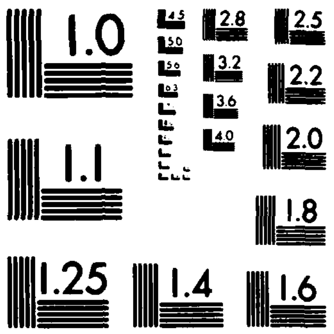
AD-A186 418 PLANNING AND TEACHING COMPLIANT MOTION STRATEGIES(U) 1/3
MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL
INTELLIGENCE LAB S J BUCKLEY JAN 87 AI-TR-936
UNCLASSIFIED N00014-85-K-0124 F/G 12/9 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

(12)

Technical Report 936

Planning and Teaching Compliant Motion Strategies

Stephen J. Buckley

MIT Artificial Intelligence Laboratory

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AI-TR-936	2. GOVT ACCESSION NO. AD-A186 418	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Planning and Teaching Compliant Motion Strategies		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Stephen John Buckley		8. CONTRACT OR GRANT NUMBER(s) N00014-85-K-0124
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE January 1987
		13. NUMBER OF PAGES 199
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Distribution is unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES None		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) motion planning, mechanical assembly, parts mating, robotics, compliance, guiding		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) See reverse side.		

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

This thesis presents a new high level robot programming system. The programming system ~~can be~~ used to construct strategies consisting of *compliant motions*, in which a moving robot slides along obstacles in its environment. The programming system is ~~re-ferred to as~~ high level because the user is spared of many robot-level details, such as the specification of conditional tests, motion termination conditions, and compliance parameters. ~~Instead~~, the user specifies task-level information, including a geometric model of the robot and its environment. The user may ~~also~~ have to specify some suggested motions.

There are two main system components. The first ~~component~~ is an interactive teaching system which accepts motion commands from a user and attempts to build a compliant motion strategy using the specified motions as building blocks. The second ~~component~~ is an autonomous compliant motion planner, which is intended to spare the user from dealing with "simple" problems. The planner simplifies the representation of the environment by decomposing the configuration space of the robot into a finite state space, whose states are vertices, edges, faces, and combinations thereof. States are linked to each other by arcs, which represent reliable compliant motions. Using best first search, states are expanded until a strategy is found from the start state to a goal state. This component represents one of the first implemented compliant motion planners.

The programming system has been implemented on a Symbolics 3600 computer, and tested on several examples. One of the resulting compliant motion strategies was successfully executed on an IBM 7565 robot manipulator.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

ARTIFICIAL INTELLIGENCE LABORATORY

Planning and Teaching Compliant Motion Strategies

Stephen John Buckley

This report was also submitted as a thesis to the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology on January 9, 1987, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Acknowledgments: This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract N00014-85-K-0124, in part by General Motors Research Laboratories, under Contract 50-10, and in part by the National Science Foundation, under contract 8451218-DMC. Support for the author was provided in part by the IBM Corporation.

Planning and Teaching Compliant Motion Strategies

by
Stephen John Buckley

Submitted to the Department of Electrical Engineering and Computer Science on January 9, 1987 in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

This thesis presents a new high level robot programming system. The programming system can be used to construct strategies consisting of *compliant motions*, in which a moving robot slides along obstacles in its environment. The programming system is referred to as high level because the user is spared of many robot-level details, such as the specification of conditional tests, motion termination conditions, and compliance parameters. Instead, the user specifies task-level information, including a geometric model of the robot and its environment. The user may also have to specify some suggested motions.

There are two main system components. The first component is an interactive teaching system which accepts motion commands from a user and attempts to build a compliant motion strategy using the specified motions as building blocks. The second component is an autonomous compliant motion planner, which is intended to spare the user from dealing with "simple" problems. The planner simplifies the representation of the environment by decomposing the configuration space of the robot into a finite state space, whose states are vertices, edges, faces, and combinations thereof. States are linked to each other by arcs, which represent reliable compliant motions. Using best first search, states are expanded until a strategy is found from the start state to a goal state. This component represents one of the first implemented compliant motion planners.

The programming system has been implemented on a Symbolics 3600 computer, and tested on several examples. One of the resulting compliant motion strategies was successfully executed on an IBM 7565 robot manipulator.

Thesis Supervisor: Prof. Tomás Lozano-Pérez
Title: Associate Professor of Computer Science

Acknowledgments

A large number of people contributed to this thesis. I will attempt to recall the major contributions here.

My advisor Tomás Lozano-Pérez was the guiding force behind the thesis. He provided frequent advice on the direction of the thesis, and helped with the details whenever I asked him to. He unwedged me when I was stuck. He carefully read through several drafts, returning valuable comments. He also advised me on many other aspects of academic life.

I was an employee of the IBM Corporation during the course of the thesis. I would like to thank IBM for its generous financial support. I would like to thank my manager, Paul Van Dyke, for his patience and friendship. I would also like to thank Russell Taylor of IBM for his advice and support.

I would like to thank the MIT AI Lab, directed by Patrick Winston, for providing the excellent environment and computational resources that made this thesis possible. They also extended financial support to me when IBM money ran out.

The emotional support of my family was instrumental. This thesis is dedicated to them, especially my father, Jim, my mother, Janet, my grandfather, Leslie Waggoner, and my grandmother, Gene Waggoner.

I learned about this area of research from Tomás, and from his other graduate students, John Canny, Bruce Donald, and Mike Erdmann. John, Bruce, and Mike generously helped me with many of the ideas in this thesis. They voluntarily read drafts of the thesis. Their friendship was as valuable to me as their expertise.

I often sought the wise advice of the mechanical engineers of the AI Lab. My most frequent sources were Mike Caine, Steve Gordon, Steve Eppinger, Karl Ulrich, Fred Martin, and Dave Brock. I also had a lot of fun hiking in the White Mountains with some of these guys.

The thesis readers, Rod Brooks and Eric Grimson, read drafts and provided valuable comments and suggestions.

Additional thanks go to Joe Jones and Pat O'Donnell, for helping me hack robots; to Oded Feingold, for providing thesis-writing hardware; to Emmanuel Mazer, for interesting discussions about my research; and to my officemate, Sundar Narasimhan, who has been a friend and a constant source of technical information.

And finally, I would like to thank Meg for her affection and support.

Contents

Chapter 1

Introduction

This thesis presents a new high level robot programming system. The programming system can be used to construct strategies consisting of ***compliant motions***, in which a moving robot slides along obstacles in its environment. The programming system is referred to as high level because the user is spared of many robot-level details, such as the specification of conditional tests, motion termination conditions, and compliance parameters. Instead, the user specifies task-level information, including a geometric model of the robot and its environment. The user may also have to specify some suggested motions.

The introduction is organized as follows. In Section 1.1, we present an example which illustrates the type of problem that the programming system is capable of solving. Then, in Section 1.2, we discuss the general problem that is to be solved. In Section 1.3, we present an overview of the robot programming system. Section 1.4 reviews previous robot programming systems. Section 1.5 summarizes the research contributions of the thesis. Section 1.6 previews the remainder of the thesis.

1.1 An Example

At this point, we present an example to give the reader an idea of the type of problem that the programming system is capable of solving. The example that we have chosen is solvable by the programming system without the need for suggested motions from the user.

Figure 1.1 depicts a three-dimensional T-shaped part, in the grasp of a robot. [1] An obstacle is shown which contains a hole with an adjoining slot. First, the T-shape is to be inserted into the hole. Then, the shaft of the T-shape is to be slid into the adjoining slot. We will assume that the robot can translate in three dimensions, but cannot rotate.

Finding a solution to this problem is complicated by uncertainty. There is uncertainty in the position and force sensing of the robot. There is also uncertainty in controlling the position and velocity of the robot. These uncertainties are typical of those that are present in the real world. Our programming system requires that

[1] This example was inspired by a similar example of Valade [1985].

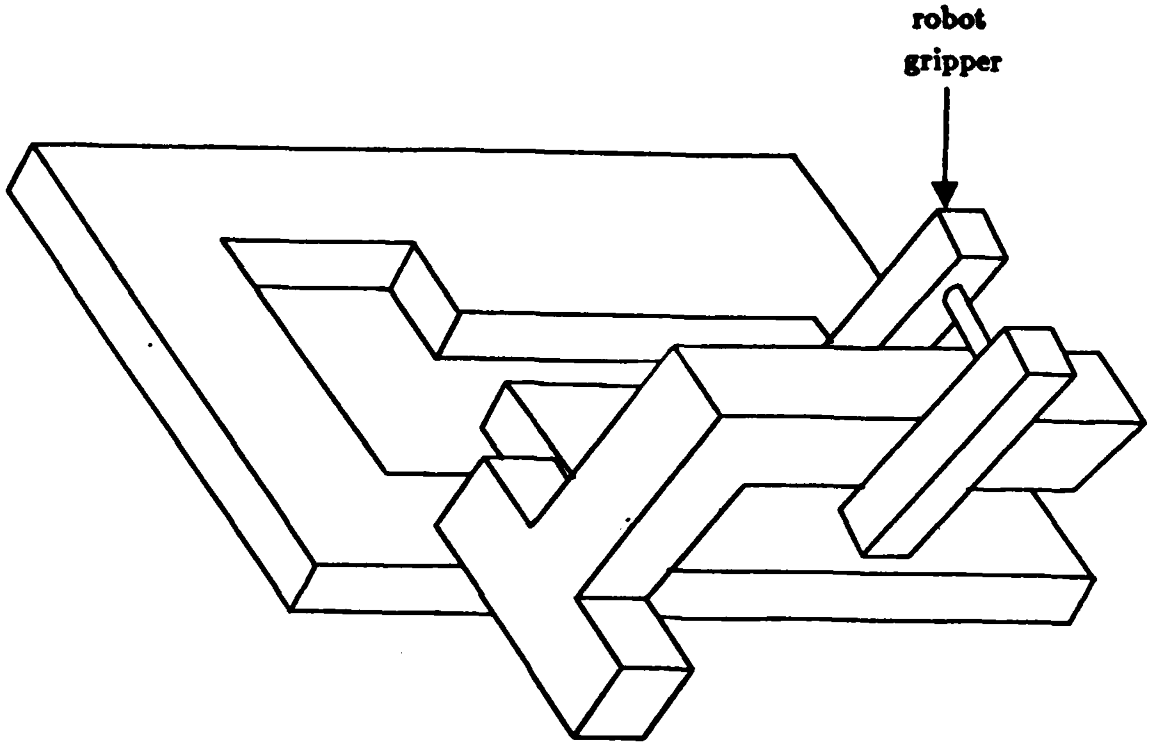


Figure 1.1: An insertion task. A three-dimensional T-shaped part is in the grasp of a robot. First, the T-shape is to be inserted into the hole of the obstacle. Then, the shaft of the T-shape is to be slid into the slot which adjoins the hole.

these uncertainties are bounded. The bounds represent worst case conditions under which a compliant motion strategy must succeed.

Let **r** be an arbitrarily chosen reference point on the robot. Consider the positions which **r** can take without causing a collision between the T-shape and the obstacle. Each face of the obstacle imposes a constraint on the free motion of **r**. These constraints are represented explicitly as surfaces in Figure 1.2. The explicit constraints consist of a sequence of two holes. The first hole is entered from a horizontal surface. This surface represents the constraints imposed by the topmost faces of the obstacle. The first hole represents the constraints on the T-shape while it is inserted into the hole in the obstacle. The second hole follows from an intermediate chamber at the bottom of the first hole. The chamber represents the additional play in the T-shape once it has fully entered the hole in the obstacle. The second hole represents the constraints on the shaft of the T-shape while it is sliding into the slot of the obstacle. Note that the geometry of the robot arm does not impose any constraints on the T-shape insertion.

The constraints of Figure 1.2 comprise a representation of the task geometry that is equivalent to that of Figure 1.1, but more explicit. This representation is called the *configuration space representation* [Arnold 1978, Udupa 1977, Lozano-Pérez 1983a]. In the new representation, we can think of the robot simply as the point **r**.

Assume that the programming system is given a configuration space model of the task geometry. Initially, the T-shape is in contact with the top of the obstacle, laterally aligned with the hole. The configuration space representation of this start region is shown in Figure 1.2. The goal region is the bottom face of the second hole, as shown in Figure 1.2. The programming system is to compute a motion strategy which moves the T-shape from the start region to the goal region despite bounded sensing and control errors in the robot. The programming system returns a strategy which consists of a sequence of two commanded compliant motions. Figure 1.3 shows the first commanded motion. The black polyhedron shown in the figure contains a set of commanded positions. The robot is to aim for any point in the polyhedron. The programming system has commanded this motion so that the robot will reach and stop on a face of the first hole, as shown in the figure. Friction between the T-shape and the hole face will cause it to stop there. The black polyhedron is behind and more narrow than the stopping region to account for possible trajectory errors. The stopping region then becomes the start region for the second commanded motion. Figure 1.4 shows the second commanded motion. If the robot aims for any commanded position in the black polyhedron shown in the figure, then the robot will enter the second hole, slide along the lower side of the hole, and stop in the goal region.

1.2 Problem Description

In this section, we describe the general problem that is to be solved. Section 1.2.1 describes the structure of a general motion planning problem. Section 1.2.2 explains

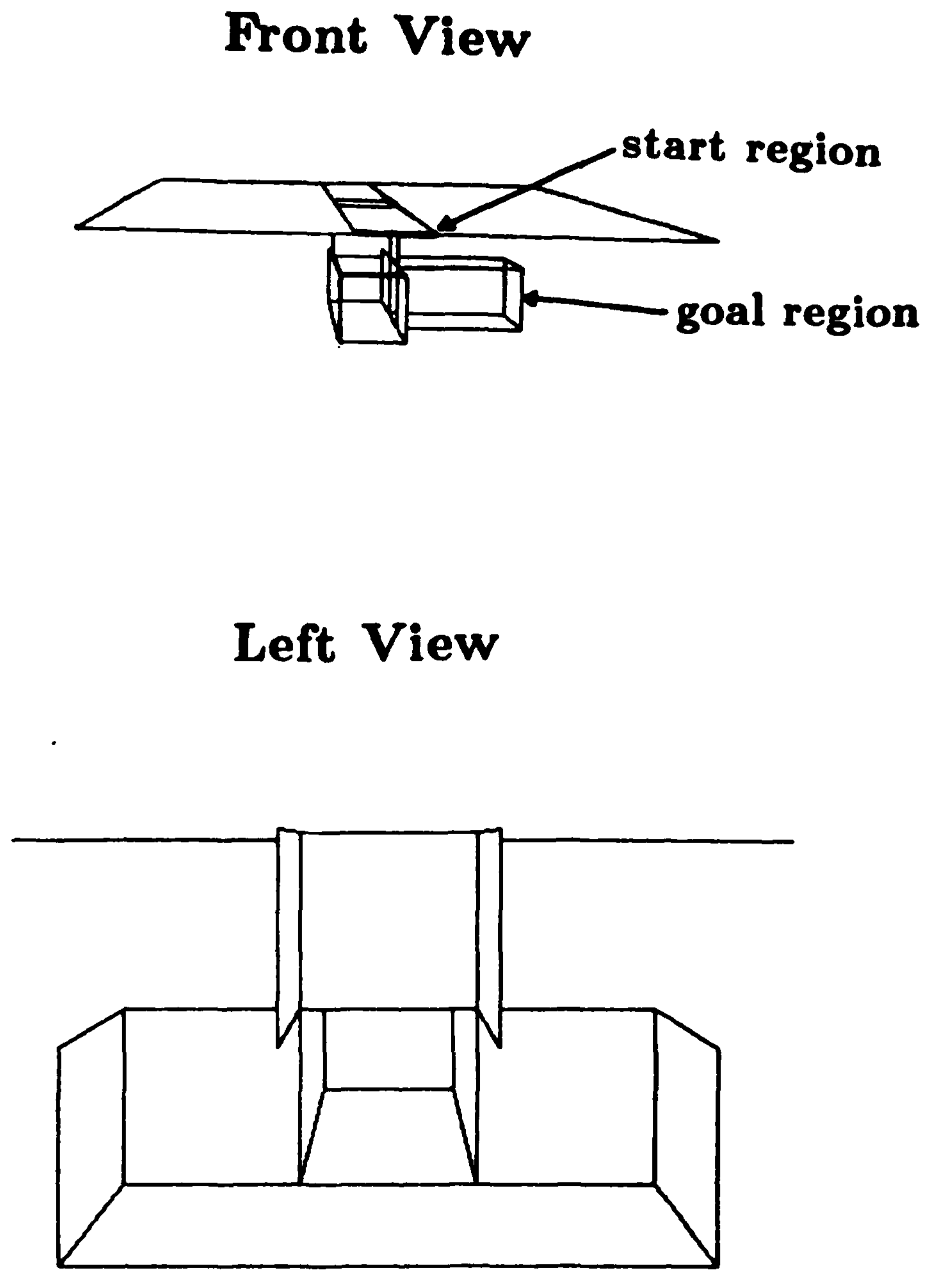
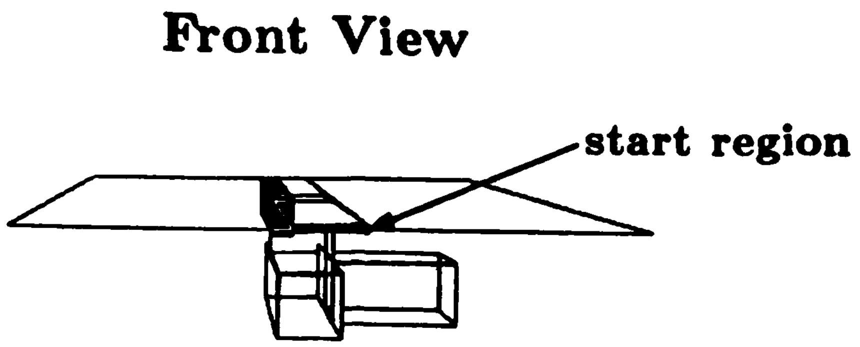


Figure 1.2: Front and left views of the configuration space representation of the T-shape insertion. The start region is an edge on the surface above the holes. The goal region is the bottom face of the second hole.



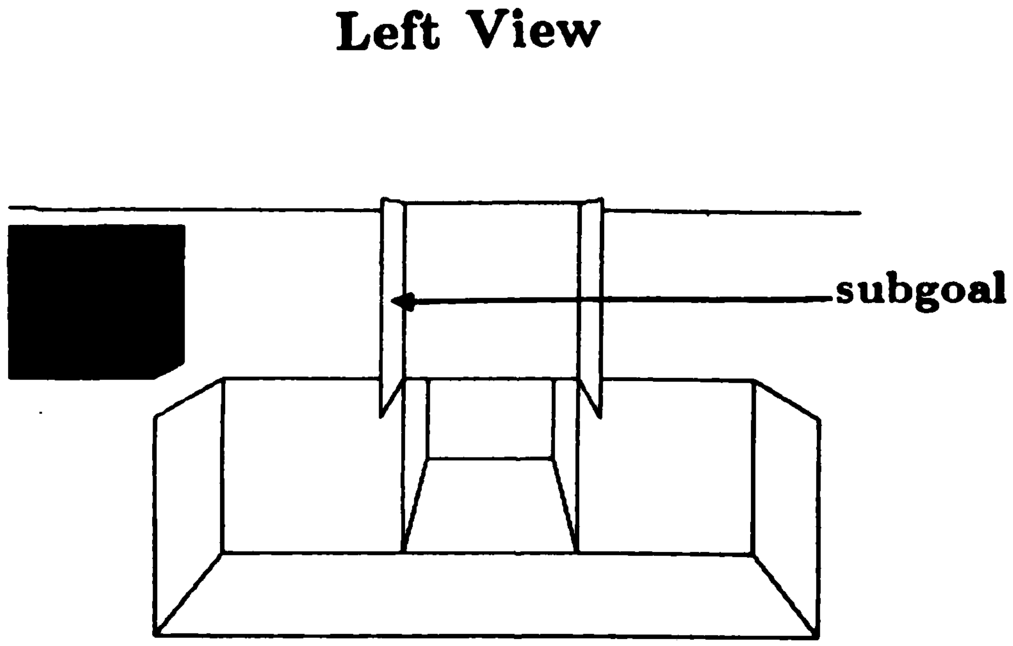


Figure 1.3: Front and left views of the first commanded motion. The start region is an edge on the surface above the holes. The subgoal for the motion is a face of the first hole, as shown. To attain the subgoal, the robot should aim for any commanded position in the black polyhedron.

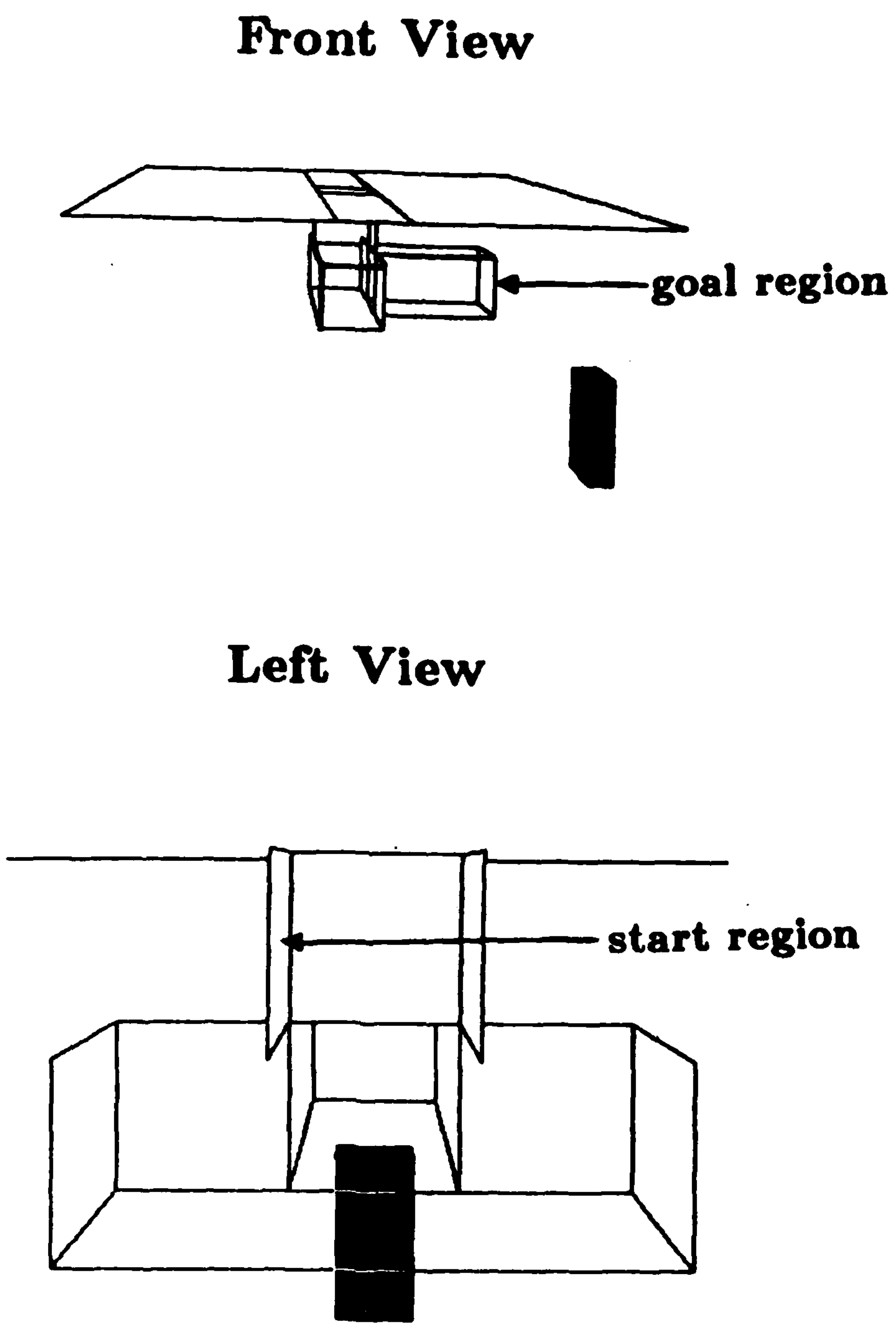


Figure 1.4: Front and left views of the second commanded motion. The start region is a face of the first hole, as shown. The goal region is the bottom of the second hole. To attain the goal, the robot should aim for any commanded position in the black polyhedron.

why we have chosen to focus on compliant motions in our solution. Section 1.2.3 describes how compliant motions can be combined into motion strategies, which represent the output of the problem. Section 1.2.4 describes the assumptions that we will make about the capabilities of the robot. Section 1.2.5 explains the relationship of the problem to the overall task of mechanical assembly.

1.2.1 Motion Planning

The programming system attempts to solve a *motion planning problem*. The input to a motion planning problem includes a geometric model of the robot, and of the environment in which it is to operate. The environment typically includes workpieces, feeders, and fixtures. Chapter 2 describes the details of the geometric models that we will employ.

The input also includes a start region, which contains possible initial positions of the robot, and a goal region, where the robot is to be placed. For simplicity, we will limit start and goal regions to positions in which the robot is in contact with the environment.

1.2.2 Compliant Motions

The framework of the robot programming system is intended for a general class of robot motions. However, this thesis will focus on a class of compliant motions. We have chosen compliant motions as a testbed for the new programming system for the following reasons:

1. Parts which are to be mated must come in contact at some point. Compliant motions are necessary at that point.

2. It is often useful to bring parts in contact as early as possible, to reduce the relative positioning uncertainty. The environment can often be used as a geometric *guide* to a goal region. (See below.)

3. It is very difficult for a human programmer to specify compliant motions. To specify a collision-free motion, a programmer must specify a position or a directional velocity. These are easy enough to specify by guiding the robot. But to specify a compliant motion, the relationship between the motion of the robot and the reaction force of the environment must somehow be specified. We need a programming system that computes compliant motion specifications based on task requirements and position or velocity specifications.

4. The success of a free space motion strategy with control uncertainty can be verified by a human programmer *a priori* by measuring the clearance of the commanded trajectories from obstacle surfaces. In the case of compliant motions, however, the success depends on factors which are more difficult to measure, such as the angle between an applied force and a surface (with uncertainty), which determines whether a robot will stick or slide on the surface. Programmers need a computational tool to verify compliant motion strategies.

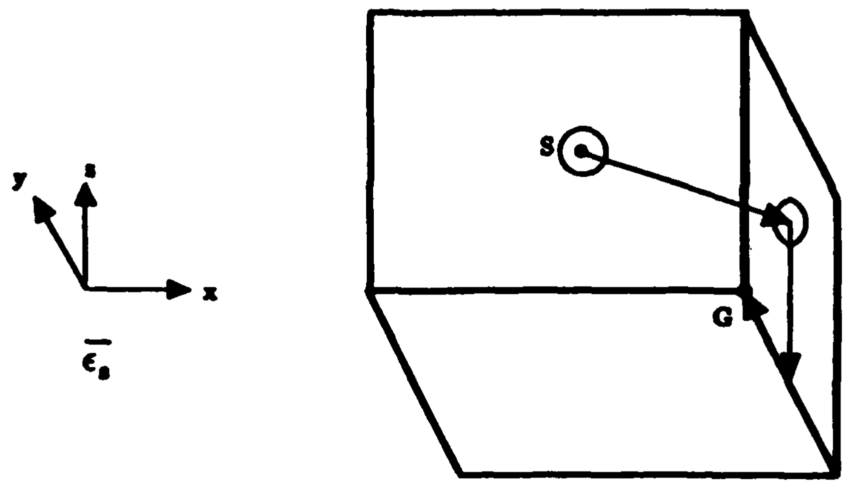


Figure 1.5: The goal G is a concave corner. Initially, the robot is known to be within a distance ϵ_s of the free space position s. First, it eliminates uncertainty in the x direction by moving to the vertical face on the right. Then, it eliminates uncertainty in the z direction by sliding along the face to the bottom edge. Finally, it eliminates uncertainty in the y direction by sliding along the edge to the corner.

The compliant motions that we are interested in must start and stop in contact with the environment. They may pass through free space to get from one surface to another. The details of their specification are given in Chapter 3.

The Environment as a Geometric Guide

When a part to be inserted is brought in contact with a surface, its degrees of freedom are reduced by at least one. If the contact surface is known, then the part's position relative to the surface is known with certainty in at least one dimension. When the part is brought in contact with k known surfaces, its position relative to the surfaces is known in at least k dimensions. For example, for a three dimensional Cartesian robot which can only translate, a part to be inserted has three degrees of freedom. If the part is brought in contact with a face of the environment, then its degrees of freedom are reduced to two, and its position relative to the face in the third dimension is known precisely. Contact with two surfaces reduces its degrees of freedom to one. Under this type of contact, the part may be able to slide along its only degree of freedom to a goal region. In this case, the environment acts as a geometric guide to the goal. Figure 1.5 shows an example of this.

The environment does not always provide a convenient geometric guide to a goal region. For example, consider the insertion task of Section 1.1. The top surface of the configuration space environment provides a vertical guide to the first hole, but there are no horizontal guides to ensure that the T-shape slides into the hole rather than past it. The programming system was able to solve this problem successfully because the positioning uncertainty of the robot was less than the hole clearance. [2]

This is not to say that the example of Section 1.1 was useless. The example demonstrates the feasibility of a planning mechanism that constructs a robust compliant motion strategy in the presence of sensing and control uncertainty. The same planning mechanism would be used even if the positioning uncertainty of the robot were greater than the hole clearance. However, for the planning mechanism to succeed under this condition, we would have to add horizontal geometric guides to the environment. This could be accomplished in the following ways:

1. Chamfers could be added to the actual hole.

2. The T-shape could be rotated, such that a smaller surface area is presented upon entry to the hole [Inoue 1974]. This would increase the clearance between the T-shape and the hole on entry. The T-shape would then have to be compliantly rotated back into alignment as it penetrates the hole. This type of insertion strategy creates an artificial chamfer in the configuration space hole [Lozano-Pérez, Mason, and Taylor 1984].

For simplicity, the use of rotation is largely ignored in this thesis. Rotations create technical complications that translations do not. The thesis focuses on the basic issues in planning compliant motions in the presence of sensing and control uncertainty. Initial feasibility will be established. Extensions to rotations are then discussed in Section 7.4.

1.2.3 Motion Strategies

The programming system is to synthesize a compliant motion strategy that is guaranteed to transport the robot from any position in the start region to some position in the goal region. A strategy may require that the robot stop in intermediate contact positions, as we saw in the example of Section 1.1. A strategy may contain conditional tests, using sensors to determine the next commanded motion. A strategy should work in the presence of bounded sensing and control errors. Figure 1.6 shows an example of a compliant motion strategy. As indicated in the figure, a strategy is represented by a directed graph. We will make the simplifying assumption that strategy graphs are acyclic, i.e., they do not have loops.

1.2.4 Robot Capabilities

We will make the following assumptions about the robot:

- The robot is a three-dimensional Cartesian robot which can translate but cannot rotate.

- The robot is capable of executing both position and velocity commands.

[2]We will define the *clearance* between a part and a hole to be the minimum distance between the part and the hole if the part is centered in the hole.

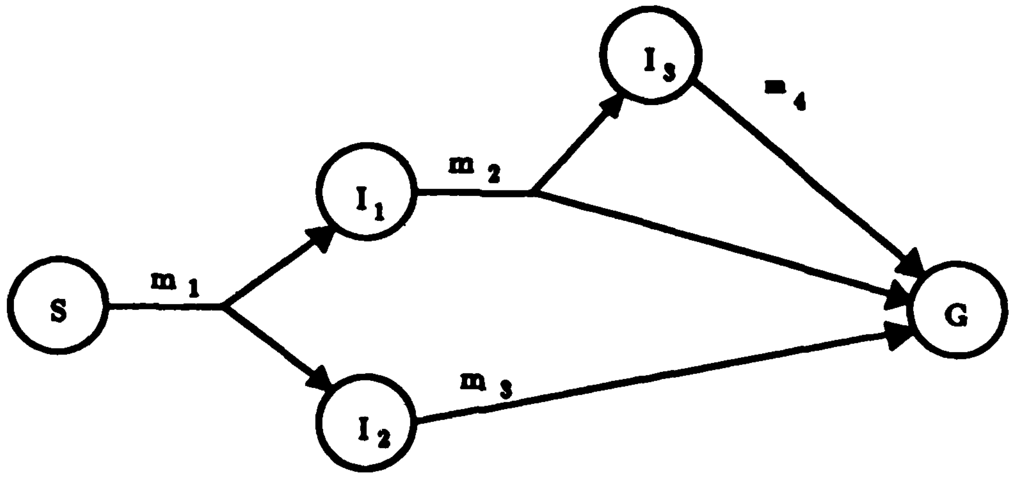


Figure 1.6: A directed graph representing a compliant motion strategy. A node in the graph represents a set of contact positions of the robot. An arc represents a commanded motion. The robot starts in set S. After issuing motion m_1, it tests whether its position and force sensors are consistent with the sets I_1 or I_2. If the sensors are consistent with I_1, then the robot issues motion m_2 next, else it issues m_3. If the strategy is reliable, then the robot eventually stops in the goal set G, despite bounded sensing and control errors.

- The robot is equipped with three-dimensional position and force sensors.

It should be noted that rotations are not always necessary. If parts are designed with chamfers, and accurately fed to a Cartesian robot such as the IBM 7565 [Taylor 1983] or the MIT Precision Assembly Robot [Vaaler and Seering 1985], then we might be able to apply our programming system directly to industrial tasks.

1.2.5 Relationship to Mechanical Assembly

In a mechanical assembly problem, a product is to be assembled by a robot from its component parts. A typical mechanical assembly task can be expressed as a sequence of motion planning problems. For example, consider Figure 1.7, in which three planar parts are to be stacked inside each other. The subtasks that the robot must solve are:

1. Grasp part 2.
2. Insert part 2 into part 3.
3. Release part 2.
4. Grasp part 1.
5. Insert part 1 into part 2.

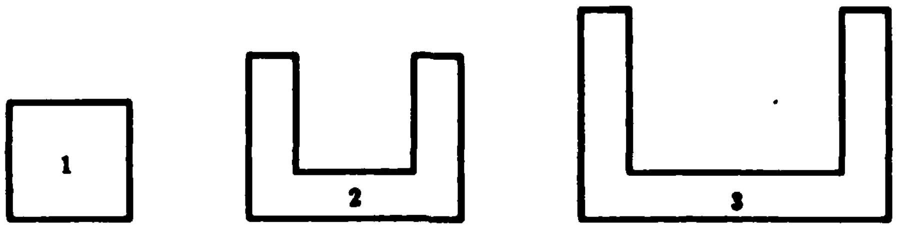


Figure 1.7: A simple planar assembly problem. First, part 2 is to be inserted into part 3. Then, part 1 is to be inserted into part 2.

6. Release part 1.

The start region of each subtask is determined by the goal region of the previous subtask. Each subtask is a motion planning problem, including grasping and releasing. One complication in formulating grasping or releasing as a motion planning problem is that the end effector of the robot adds degrees of freedom to the problem. For example, consider the problem of grasping the T-shape of Figure 1.1 with a parallel jaw gripper. We can formulate this as a motion planning problem with four degrees of freedom, three for the translational axes of the robot, and one for the gripper. Note that a more sophisticated robot hand would add even more degrees of freedom.

1.3 Overview of the Robot Programming System

Figure 1.8 shows an operational view of the robot programming system. The user submits a problem to the system, consisting of an environment, a start region, and a goal region. The system then attempts to solve the problem autonomously. If it succeeds, then the resulting compliant motion strategy is returned. Otherwise, user interaction is required.

1.3.1 Modeling Robot Motions

A user needs an abstract model for specifying compliant motions. In Chapter 3, we review a number of currently existing models. We then focus on two particular models which fit our needs, the *generalized spring* model, and the *generalized damper* model. Under the generalized spring model, the user specifies commanded positions at which the robot is to aim. Under the generalized damper model, the user specifies commanded velocities which define directions in which the robot is to move. Under each of these models, a representation is developed for bounded control uncertainty, in both free space and in contact. This representation of control uncertainty allows trajectories to be simulated, which in turn facilitates autonomous motion planning.

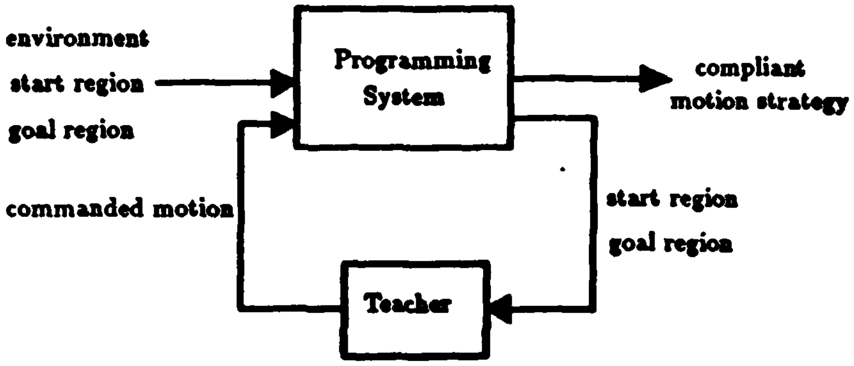


Figure 1.8: An operational view of the robot programming system. The user submits a problem to the system, consisting of an environment, a start region, and a goal region, the system then attempts to solve the problem autonomously. If it succeeds, then the resulting compliant motion strategy is returned. Otherwise, the user is asked to supply suggested motions, which can be either commanded positions or velocities.

1.3.2 The Teaching System

When user interaction is required, the programming system displays the start and goal regions graphically, and prompts the user. The user then submits a commanded motion m, which can be either a commanded position or velocity. In principle, the commanded motion could be entered by guiding the robot, or with a light pen. For our initial implementation, the commanded motion was simply typed in. The commanded motion should be one which the user hopes will reach the goal region reliably from the start region. If this is impossible, then the user should submit what is hoped will be the last motion in a reliable sequence of motions. Minimally, the motion should be one which reliably reaches the goal region from *somewhere*.

The system then computes a set R of contact positions from which the goal region can be reliably reached and terminated in via m. R is called a *pre-image* of the goal region under m [Lozano-Pérez, Mason, and Taylor 1984]. Pre-images can be computed for certain termination conditions by geometric backprojection from a subset of the goal region, called a *backprojection base* [Erdmann 1984, 1986]. A backprojection base contains points that can be recognized as goal points using sensing. Backprojection computes points which are guaranteed to reach the backprojection base despite bounded sensing and control uncertainty. In our teaching system, we will extend Erdmann's algorithm to three Euclidean dimensions, under both the generalized damper and spring trajectory models, for the case of compliant motions.

Once the pre-image R has been computed, it is stored in a table, along with m and the subset of the goal region that can be reached from R. R is now considered to be *solved*, and is added to the goal region. If a subset S' of the start region is recognizably contained in R despite sensing uncertainty, then S' is solved, and can be removed from the start region. If the new start region is empty, then the problem is solved. Otherwise, the system attempts to solve the new problem autonomously. If this fails, then the system displays the new start and goal regions, and user interaction continues.

By iteratively reducing the size of the start region, and increasing the size of the goal region, it is hoped that the user and system can together converge on a successful strategy. However, this is not guaranteed. For instance, if no successful strategy exists under the given error bounds, then the iterative process will fail.

The interaction between the programming system and the user defines a new method for teaching motion strategies to robots. The main advantage of this teaching system is that the user is spared the difficult tasks of generating conditional sensor tests and motion termination conditions, and testing the program to ensure that it works under bounded sensing and control errors.

1.3.3 Autonomous Planning

One might wonder why the teaching system is necessary, given an autonomous compliant motion planning capability. The problem of planning compliant motions with uncertainty is an instance of the general problem of planning motions with uncertainty. Recently, Canny and Reif [1986] showed that the general problem is exponential time hard. The general problem may actually be unsolvable [Erdmann 1984]. In spite of these theoretical constraints, our goal was to implement a working planner. The result is an implemented planner which does not always succeed when a solution exists. Experimentation has indicated that the planner can solve a useful class of problems nevertheless.

An *atom* is a set of contiguous configurations which share a common set of possible reaction forces. Any face, edge, or vertex from the configuration space environment is an atom. All of free space is an atom. The planner characterizes the state of the robot as a collection of atoms where the robot might be located, under bounded sensing and control uncertainty.

The planner operates by repeatedly choosing a state, and constructing arcs which connect the state to other states. Arcs represent reliable motions between states. Arc construction proceeds from state to state until a successful compliant motion strategy has been constructed from the start state to a goal state.

To expand the arcs of a state, the planner computes all possible commanded motions from the state, grouping them into sets called *weak arcs*, which have a common target atom. Weak arcs are then composed into reliable *strong arcs*, which share a common set of target atoms. Subsets of the target atoms of a strong arc represent target states that the robot might reach if it follows the arc. On execution, the robot decides which target state it has actually reached using position and force sensing.

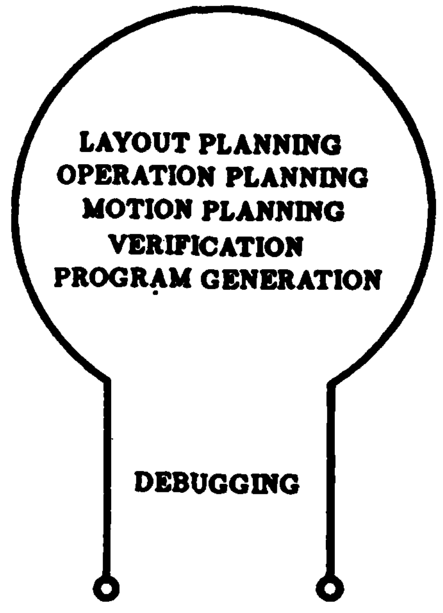


Figure 1.9: Typical steps in the design of a robot application. A robot programmer often spends two thirds of the total development time debugging.

1.4 Previous Robot Programming Systems

Figure 1.9 shows typical steps in the design of a robot application. In the *layout planning* stage, support hardware is designed. Then, in the *operation planning* stage, the overall task is broken into simpler subtasks. Typical subtasks are grasping a part, transporting a part to another location without collision (*gross motion*), mating a part with another part (compliant motion), and releasing a part.

Next, in the *motion planning* stage, a skeletal motion strategy for each subtask is generated, usually in the form of a procedure. In the *verification* stage, possible errors in the strategy are computed, using estimates of uncertainties. Then, in the *strategy synthesis* stage, a more robust motion strategy is generated which detects and corrects predicted errors. Finally, in the *debugging* stage, the robot procedure is tested and debugged. A bottleneck is shown at this stage, because two thirds of the development time is often devoted to this activity in practice [Will 1981].

The robot programming system plays a critical role in this development process. Different systems provide radically different capabilities. They can be classified into three general categories: guiding systems, robot-level languages, and task planning systems [Lozano-Pérez 1983b, Bonner and Shin 1982]. These categories are discussed below.

1.4.1 Guiding Systems

In a *guiding system*, or *teaching system*, the user leads the robot through the motions to be performed, either by physically grasping the robot, or by manipulating a button box or joystick. Most guiding systems produce only decision-free point-to-point motion strategies. The specification of conditional tests and sensing is difficult or impossible. The narrow capabilities of these systems limit their utility to a small set of applications, including painting and welding. See Lozano-Pérez [1983b] for a survey of guiding systems of this nature.

Research efforts have concentrated on giving guiding systems more function, but without much success. Buttons were added to teach pendants to do things like switch between coordinate systems, adjust the speed, and save and recall taught positions. Buttons were also added to perform special canned motion strategies. For example, the FUNKY system [Grossman 1977] had a GRASP routine that used finger sensors to regulate squeezing forces. In addition, FUNKY had a CENTER command which centered the gripper over an object, and a SEARCH command which performed a *guarded move* (a motion terminated by detected contact). The POINTY system [Grossman and Taylor 1978] allowed object models to be defined by touching objects with the manipulator while describing the objects interactively. The resulting geometric models could then be referred to by a robot-level program written in the AL language [Finkel et al 1974].

A number of subsequent systems allowed guarded moves to be taught. In XPROBE [Summers and Grossman 1984], the user informs the system before each motion whether there are guard conditions for the motion. The system monitors the force sensors during the motion and at its completion (which is also signalled by the user) determines whether the guard condition is being used to terminate the motion normally or to detect unexpected contact. A similar system, TRIG [McLaughlin 1982], allows guarded moves to be taught, which result in a conditional test that branches one of two ways depending on whether contact occurred.

A few guiding systems have attempted to generate conditional tests. In these systems, the user must explicitly inform the system that a decision point has been reached. Both FUNKY and TRIG fall into this category; conditional tests based on sensor values are generated when a guarded move is taught.

Asada [1979] investigated the teaching of compliant motions for a three-fingered hand. The teacher guides the robot to the neighborhood of the contact location using position control. Then, the teacher presses or pulls the robot fingers until the desired contact forces are obtained. The system then computes the finger stiffness parameters required to replay the motion. Errors in the calculations are (hopefully) detected by replaying the compliant motion for the user's approval.

Hirzinger and Landzettel [1985] implemented a similar system. A motion sequence is presented by the teacher using a joystick. During the guided motion sequence, the system records sensed positions and forces at regular intervals. Each recorded interval generates a commanded motion. The system derives a commanded position for each interval which satisfies the equation of a generalized spring trajectory. On playback, the user can modify the trajectory with guided forces, to correct errors in the original calculations.

All of the above systems are limited by the following characteristics:

1. The procedures that are generated by the system are not guaranteed to work

under uncertainty. The debugging bottleneck remains.

2. The user is responsible for planning the decisions of the assembly. The system has no ability to infer conditional tests.

Inference of procedures from example executions is an instance of *learning*, or more specifically, ***procedural acquisition***. A few recent papers from this area have applied procedural acquisition to robot procedures. One system, by Selfridge and Levas [1983], uses a knowledge-based approach. Guided input is matched with plans in a planning library. If a portion of the input can be matched with a stored plan, then the input can be used to fill in details that were left as parameters in the plan. If not, then a new plan can be stored. This system considers only very simple examples, and does not address conditional tests, compliant motions, or uncertainty.

Segre and DeJong used the explanation-based learning paradigm to generalize robot procedures. The input consists of a geometric model of the parts, and a sequence of collision-free motions, such as grasps, translations, rotations, and releases. Subsequences of input motions are matched against stored plans called *task schemata*. Task schemata contain commanded motions and calls to other task schemata. At the end, after subsequences have been matched, a new task schema is stored. Matching of motions to task schemata is enhanced by two other types of matching, of task parts to *physical object schemata*, and of assembled parts to *mechanism schemata*. Physical object schemata are simplified geometric models of parts. Mechanism schemata are kinematic descriptions of the relationship between assembled parts. Matching to these other types of schemata gives the system a more task-oriented basis for matching. Conditional tests, compliant motions, and uncertainty are ignored.

Dufay and Latombe [1984] examined robot procedural generalization with conditional tests and compliant motions. Their system consists of two modules, a planner and an execution monitor. The planner generates a robot procedure from a set of planning rules, and the execution monitor executes the procedure, examining force and position feedback to see if it worked. If the procedure did not work, the planner is notified and must produce an alternative plan. This process continues until several successful procedures have been executed. These procedures are then generalized into a single procedure which contains conditional tests and guarded moves. Note that this system would be a teaching system if the plans came from a human operator rather than a planning system. One criticism of this system is that it lacks a general geometric reasoning capability. The interpretation of sensory data is performed on a task basis by interpretation rules supplied with the planning system. A more fundamental problem lies in the overall strategy for generating procedures:

1. It tests suggested robot procedures by actually executing them. Each test run is time consuming, and the application hardware has to be physically set up before each test run. Physical testing may also be inadvisable. Assembly applications consist of expensive robots, feeders, and fixtures, which should not be subjected to unnecessary collisions.

2. In general, after a finite number of test runs, the resulting procedure is not guaranteed to work under uncertainty. For a given application, there is no way to know how many test runs would be required before a procedure is reliable. The debugging bottleneck is still present.

The generalization mechanism introduced by Dufay and Latombe consists of merging redundant execution traces for the same part operation into a single procedure with conditional tests and loops to take care of the uncertainty-caused variations. Interestingly, it turns out that Dufay and Latombe's generalization framework with extensions can be used to solve an important higher level problem: the construction of a procedure which performs *multiple* part operations or which performs a common operation on *different types* of parts. This thesis does not address that problem. For example, suppose some blocks are to be stacked. A human programmer would implement this as a loop surrounding a procedure to stack a single block. The goal position for the stacking procedure would be written as an expression over a variable which varies with the block that is to be stacked. This thesis only addresses motions where the goal region is fixed. The higher-level problem would be to generalize a loop from procedures produced for individual blocks.

A recent system called NODDY [Andreae 1985] is concerned with this generalization problem. NODDY is a domain-independent procedural generalization system which has been applied to motion planning. NODDY ignores the kind of low-level uncertainty that this thesis concentrates on. However, given outputs from the system described here (procedures which can perform a single part operation with a fixed goal region), NODDY would in principle be able to generalize a procedure which performs a series of part operations.

1.4.2 Robot-level Languages

In a *robot-level language*, robot motions are controlled by commands from a programming language such as AL [Finkel et al 1974], VAL [Unimation 1980], and AML [Taylor et al 1982]. Modern programming languages allow a trained programmer to specify parameterized robot motion and sensing commands, as well as standard software functions such as data abstraction, procedural abstraction, conditional branching, arithmetic, database management, and communication with other computers. See Lozano-Pérez [1983b] and Buckley and Collins [1985] for discussions and examples of robot-level programming.

Robot-level languages have exactly the opposite characteristics as guiding systems. Although they can be very powerful, the more powerful they get, the harder they are to use. From my experience, an industrial robot programming application can take six months or more to design and implement. Debugging the robot procedure can involve as much as two thirds of this time.

Robot-level languages are inflexible because it takes so long to convert from one application to another. The conversion time is long because of the difficulty of designing and debugging a robot procedure. Many things can go wrong in a robot application, and they all have to be accounted for in the procedure. However, it is difficult to predict what will go wrong before actually trying out the procedure. This creates a bottleneck when testing and debugging a robot procedure.

1.4.3 Task Planning

The goal of *task planning* research [Lozano-Pérez 1976, Taylor 1976, Lieberman and Wesley 1977] is to relieve the user of as many implementation decisions as possible. This requires that the programming system perform many of the duties that are shown in Figure 1.9. A task planner would be both powerful and easy to use.

Task planning provides flexibility in robot programming by supplying the necessary robot functions without the associated debugging bottleneck. The idea is that a robot procedure which systematically prepares for all likely errors will not need to be debugged, thereby eliminating the bottleneck. Furthermore, if the procedure is generated by computer, then the programmer is relieved of the tedium of predicting and correcting for possible errors. However, many difficult research problems need to be solved before a computer can generate such a robust procedure.

Motion planning is the only area of task planning that has received significant attention, and it is not completely solved yet. The three key subproblems of motion planning are grasp planning, gross motion planning, and compliant motion planning.

Grasp planning involves two basic problems, finding grasp points which stably constrain a workpiece, and planning collision-free trajectories of the robot fingers to grasp points. Representative research on the first problem includes Salisbury [1982], Fearing [1984], Cutkosky [1985], Kerr [1985], and Nguyen [1986]. Representative research on the second problem includes Lozano-Pérez [1976], Laugier [1981], and Peshkin and Sanderson [1986].

Gross motion planning involves the construction of a sequence of motions that will move a robot without collision to a goal region, using a perfectly controlled robot in a perfectly known environment. This subproblem has been examined for a rigid polyhedral robot moving through a polyhedral environment. Representative research in this area includes Udupa [1977], Lozano-Pérez and Wesley [1979], Lozano-Pérez [1981], Schwartz and Sharir [1982], Lozano-Pérez [1983a], Brooks [1983], Brooks and Lozano-Pérez [1983], Donald [1984], and Lozano-Pérez [1985].

Early research in compliant motion planning involved filling in the details of partially specified procedures called *skeleton strategies* [Taylor 1976, Lozano-Pérez 1976] using actual model parameters. A basic assumption of this research was that there would be a relatively small number of basic operations in a particular domain. For example, peg-in-hole insertion is a common operation in the domain of mechanical assembly [Draper 1983]. Inoue [1974] had previously formulated a seemingly robust strategy for inserting a peg into a hole. The hope was that this strategy could be parameterized in such a way that a task planner could use it for virtually any insertion. Taylor [1976] designed skeleton strategies whose motion commands were in terms of part locations, which in turn were a function of model parameters and initial positions. In addition, strategies contained *metadecisions*, to be resolved by the task planner, which offered a choice between several substrategies based on part locations. For a particular task, part locations were computed by the task planner from model parameters and initial positions by evaluating chains of homogeneous transformations representing part relations. This evaluation was performed by numerical propagation of model parameters, initial positions, and associated uncertainties, using linear programming. Brooks [1982] designed a more robust implementation for the propagation process, using symbolic algebra. His technique resulted in backconstraints on model parameters and initial positions that ensure the success of the strategy under uncertainty. The idea of backconstraining the initial conditions of a skeleton strategy was first used by Lozano-Pérez [1976]. In his planner, after motion parameters had been instantiated in a strategy, the strategy was geometrically simulated to detect undesired collisions. These collisions were then used to constrain the initial positions of the subtask, in the hope that previous subtasks in the assembly could achieve refined goal positions. Research into skeleton strategies has received little attention of late, in light of the observation that even simple variations in task geometry require radical changes in a motion strategy [Lozano-Pérez, Mason, and Taylor 1984].

Lozano-Pérez, Mason, and Taylor [1984] took a different approach, based on the idea of pre-images. Under their framework, a sequence of motions to move the robot from a start region to the goal region would be found by backward chaining from the goal region, recursively computing pre-image regions. The resulting strategies might contain conditional tests. They did not propose an implementation for the framework. However, the framework introduced many fundamental planning concepts, including pre-images, the generalized damper trajectory model, a theory of termination conditions, and backward chaining as applied to motion planning. The teaching system of Chapter 5 is based on this framework. Mason [1984] established completeness and correctness results for the framework under several types of termination conditions.

Erdmann [1984,1986] continued the work on this framework. He showed that for certain classes of termination conditions, pre-images can be computed by geometric backprojection from a subset of the goal region. His algorithm was implemented for planar robots with rotation. A key component of his algorithm was a representation of friction in configuration spaces which describe rotation as well as translation.

The approach taken by Lozano-Pérez, Mason, and Taylor has thus far evaded implementation in full generality. Other approaches to compliant motion planning use simplifications or heuristics to make the problem more tractable. Some approaches are simply manual procedures to be followed by human programmers; others are implemented computer programs that have been tested on simulated environments. Few approaches have been extensively tested on real robots, primarily because of the unavailability of robots which can perform compliant motions. See Whitney [1985] and An [1986] for discussions of current issues in compliance control.

Manual approaches to compliant motion planning were taken by a number of researchers. Simunovic and Whitney [Simunovic 1979, Whitney 1982] developed a one-step motion strategy for inserting a cylindrical peg into a cylindrical hole. The strategy is valid for pegs and holes which meet certain parametric specifications. They assumed the generalized spring trajectory model, choosing to command the applied robot force. Their task was to compute the robot stiffness and applied force which causes the peg to enter the hole without sticking. They did this by solving a set of quasi-static force balance equations generated by geometric and frictional constraints when the peg is sliding into the hole. Ohwovoriole and Roth [1981] derived a similar one-step strategy for inserting a peg into a hole.

Caine [1985] developed a manual process for designing decision-free multistep motion strategies, in three dimensions with rotations. He assumed the force control trajectory model (Chapter 3). His method decomposes a given environment into states which characterize the type of contact between the robot and environment. Although he does not compute them explicitly, his states correspond to obstacle faces in the six dimensional configuration space that represents the possible translations and rotations of the robot. The states are manually searched for a connected sequence which the robot can slide through without breaking contact. The commanded forces which effect the sliding motions must avoid sticking along the way, and sliding to the wrong state due to control uncertainty. These constraints are implemented by intersecting sets of generalized forces, obtained by solving complex quasi-static force balance equations. In a six dimensional generalized force space, this is quite tedious. The design process was applied to the insertion of a rectangular peg into a rectangular hole. Although it is not algorithmic, Caine's design process generalizes the method of Simunovic and Whitney. The autonomous compliant motion planner described in Chapter 6 is based on many of the principles used in Caine's method.

Koutsou [1985] proposed an automatic method for finding a sequence of sliding motions of a robot in contact with an obstacle. The input includes a set of qualitative spatial relations describing the start and goal configurations of the robot. The proposed method constructs a graph whose nodes represent obstacle faces in a six-dimensional configuration space, and whose arcs represent physical connections between faces. The spatial relations are compiled into start and goal faces in the configuration space. The graph is then searched for a connected path from the start face to the goal face. She did not consider whether compliant motions would actually be possible along the resultant path. It should be noted the Donald [1984,1985] presented an earlier algorithm for computing obstacle faces in a six-dimensional configuration space.

Laugier and Theveneau [1986] are building a system which will construct and search a state graph, generating compliant motions between states. Each state describes a particular contact type between the robot and the environment, similar to Koutsou. Arcs, which represent compliant motions, are constructed by backward chaining from the goal state along heuristically chosen sliding directions. The sliding directions consist of simple directions parallel and perpendicular to contact edges, and directions chosen by expert rules. The sliding directions are extended until a new, stable contact type is encountered. The location of the new contact type becomes the start state of the compliant motion. The sliding motion is specified as a generalized damper motion which avoids sticking on the surface, and terminates when a reaction force of the goal state is encountered. At present, the system is limited to translational motions. A similar approach by Valade [1984, 1985] performs a heuristic search for translational sliding directions by analyzing concavities in the assembly parts.

Turk [1985] examined compliant motion planning in the plane without rotation. He defines a state as a region of free space, bounded by a set of configuration space edges. Arcs are constructed between states, labeled by the range of commanded velocities that are guaranteed not to stick on a bounding edge of the initial region on the way to the target region, under the generalized damper trajectory model. Motions are terminated by a combination of position, force, and time conditions. The state graph is then searched for a decision-free chain of motions which lead to the goal region. The method works best in tunnels where one region of free space naturally leads to another. It has problems in other situations, such as entering a hole from an open space.

Erdmann and Mason [1986] examined compliant motion planning in the plane with rotation. They define a state as a set of possible contact types between the robot and the environment. Defining a state as a set of possible contact types rather than a single contact type is necessary because of uncertainty. Using forward chaining from the start state, arcs are constructed between states, representing reliable sliding motions between the states. Motions are terminated by sticking. The computation of sliding motions is facilitated by Erdmann's configuration space representation of friction under rotation, as discussed above. Forward chaining ends when a goal state is found. The resulting strategy is decision-free. The planner was applied to a tray containing an Allen wrench. The Allen wrench is moved from an arbitrary position and orientation to a known position and orientation by applying a sequence of tilts to the tray.

The paper by Erdmann and Mason illustrates the close link between planning compliant robot motions and planning general pushing and sliding operations. Other research on pushing and sliding includes Mason [1982, 1986], Mani and Wilson [1985], Brost [1986], and Peshkin [1986].

Much of the compliant motion planning research assumes some type of uncertainty in the sensing and control of the robot. More difficulty is encountered when we consider errors in the geometric models of the robot and environment. Donald [1986] has begun a study of this problem.

1.5 Summary of Research Contributions

This thesis makes the following new contributions to the theory of robot programming:

1. An abstract model of a position controlled robot is developed, based on first order dynamics with the robot modeled as a damped spring. This model can be used by motion planners and human programmers to plan both collision-free and compliant motions.

2. A method for verifying the correctness of a compliant motion strategy is developed and implemented for a particular model of compliant motions.

3. A method for teaching a compliant motion strategy is developed and implemented for two models of compliant motions.

4. A method for autonomously planning a compliant motion strategy is developed and implemented for a particular model of compliant motions.

1.6 Outline of the Thesis

The rest of the thesis is organized as follows:

- Chapter 2 describes the geometric models that we will use to represent the robot, workpieces, and other obstacles in the environment.
- Chapter 3 describes the abstract models that we will use to specify robot motions.
- Chapter 4 presents a method for verifying the correctness of a compliant motion strategy. The need for a computational tool of this nature was mentioned above. Also, the concepts involved in verifying correctness are building blocks for the remainder of the thesis.
- Chapter 5 presents the details of the teaching component of the robot programming system.
- Section 6 presents the details of the planning component of the robot programming system.
- Section 7 presents ideas for future work in this area.
- Appendix A presents a method for computing visible surfaces in a three dimensional environment. This is used in the thesis to simulate commanded motions for verification and teaching.
- Appendix B contains some derivations for the generalized spring trajectory model.
- Appendix C contains a robot program which implements a peg-in-hole insertion using results from the thesis.

Chapter 2

Modeling Geometry

We will use polyhedral models to represent the geometry of the robot and its environment. The environment of a typical mechanical assembly includes workpieces, feeders, and fixtures. Many of these objects can be accurately modeled by polyhedra. Furthermore, curved objects can be approximated by polyhedral models with many faces. The accuracy of the approximation can be controlled by adjusting the number of faces in the model. Artificial surface features that are introduced by the approximation need not present a problem for a motion planner which takes uncertainty into account.

The reason for using polyhedral models is that in many cases they simplify the representation of an object. For example, interior points of a face need not be represented explicitly. Furthermore, the reaction force of every point on a face can be predicted by a single computation for the whole face. This simplifies the task of reasoning about compliant motions.

For simplicity, we will assume that the geometric models are known perfectly. Later, we will comment on the effects of geometric modeling error in Section 7.2.

2.1 Configuration Space

In section 1.1, we introduced by example a geometric representation called *configuration space* [Arnold 1978, Udupa 1977, Lozano-Pérez 1983a], in which the constraints that obstacles impose on a moving robot are explicitly represented as surfaces. In this section, we discuss this representation in more detail.

The *configuration* of a moving robot is a set of parameters which completely determines its position and orientation. The configuration space of a robot is defined as the space of possible robot configurations.

The configuration of a three-dimensional Cartesian robot that can only translate is given by three parameters, one for each dimension. Let $\mathbf{r}$ be an arbitrarily chosen reference point on the robot. We can use the coordinates of $\mathbf{r}$ to represent the configuration of the robot. The configuration space for this type of robot is isomorphic to the three dimensional Euclidean space $\Re^3$.

There are many other types of configuration spaces, depending upon the kinematics of the robot. Here are some examples:

- The configuration of a three-dimensional Cartesian robot that can both translate and rotate is given by six parameters, three for its position, and three for its orientation. The three positional parameters can be given by the coordinates of a reference point. Rotations can be represented in a number of ways, including Euler angles, roll-pitch-yaw coordinates, cylindrical coordinates, and unit quaternions [Paul 1981, Canny 1986]. When rotations are represented by unit quaternions, the space of three dimensional rotations is isomorphic to the special orthogonal group of three parameters, $SO(3)$. The configuration space of the robot is then isomorphic to the product space $\Re^3 \times SO(3)$.

- The configuration of a robot with six rotary joints can be specified by six parameters, one for each joint angle. The configuration space of each joint can be represented by a subset of the circle S^1. Each subset is limited by kinematic constraints on joint travel. The configuration space of the robot can then be represented by a subspace of the product space $S^1 \times S^1 \times S^1 \times S^1 \times S^1 \times S^1$, otherwise known as the six-torus.

There are a number of technical difficulties that arise when dealing with a configuration space of dimension higher than three, or with a non-Euclidean configuration space such as a rotation space. We will discuss some of these difficulties in Chapter 7. For now, we will assume that the configuration space of the robot is the three dimensional Euclidean space $\Re^3$.

2.2 Contact Space

Consider the specification of the start region S. For simplicity, S is limited to configurations which correspond to contact between the robot and the environment. S is a subset of *contact space*, the set of all configurations in which the robot is in contact. Figure 2.1 shows the contact space for a planar example. For a planar Cartesian robot which can only translate, contact space consists of a set of edges. Each edge is generated by one of the following combinations:

- a vertex of the robot and an edge of the environment.

- an edge of the robot and a vertex of the environment.

The vertices of the planar contact space correspond to configurations in which the robot is in contact with at least two environment surfaces.

For a three dimensional robot which can only translate, the contact space consists of a set of faces. Each face is generated by one of the following combinations:

- a vertex of the robot and a face of the environment.

- a face of the robot and a vertex of the environment.

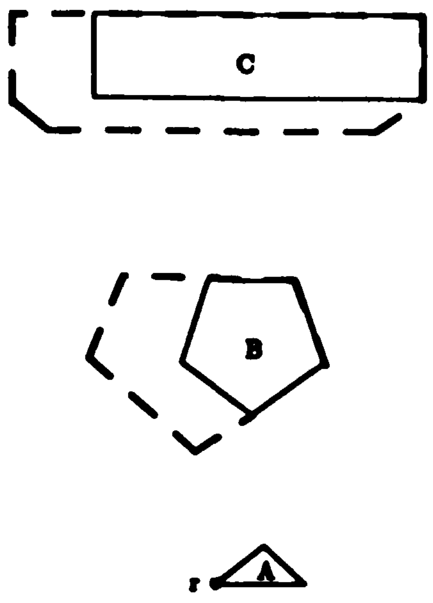


Figure 2.1: The reference point **r** is chosen as a vertex of the robot A. The dotted lines show the contact space of A with respect to the obstacles B and C.

- an edge of the robot and an edge of the environment.

Figure 2.2 shows these combinations. The edges of contact space correspond to configurations in which the robot is in contact with at least two environment surfaces. The vertices of contact space correspond to configurations in which the robot is in contact with at least three environment surfaces.

Our start and goal regions are represented as collections of polygons from contact space. But this is not the only use of contact space. Contact space is an explicit representation of the constraints that obstacles impose on a moving robot. In other words, contact space bounds *free space*, the free configurations of the robot. To plan collision-free trajectories, a motion planner must pick trajectories which remain in free space. To plan sliding trajectories, a motion planner must pick trajectories which remain in contact space. When we use contact space in this way, we are essentially shrinking the robot to its reference point **r**, and growing the environment such that it is bounded by contact space. This representation of a robot and its environment is called the *configuration space representation*. For example, the configuration space representation of Figure 2.1 is obtained by replacing the robot by its reference point **r**, and bounding the obstacles B and C by the indicated contact space surfaces.

In Section 1.1, we showed a three dimensional example of the configuration space representation. Figure 1.2 showed the configuration space representation of the motion planning problem depicted in Figure 1.1.

Lozano-Pérez [1983] and Donald [1984] developed algorithms for computing a configuration space representation of a Cartesian robot, represented as a polyhedron in a polyhedral environment. For the remainder of this thesis, we will simply assume

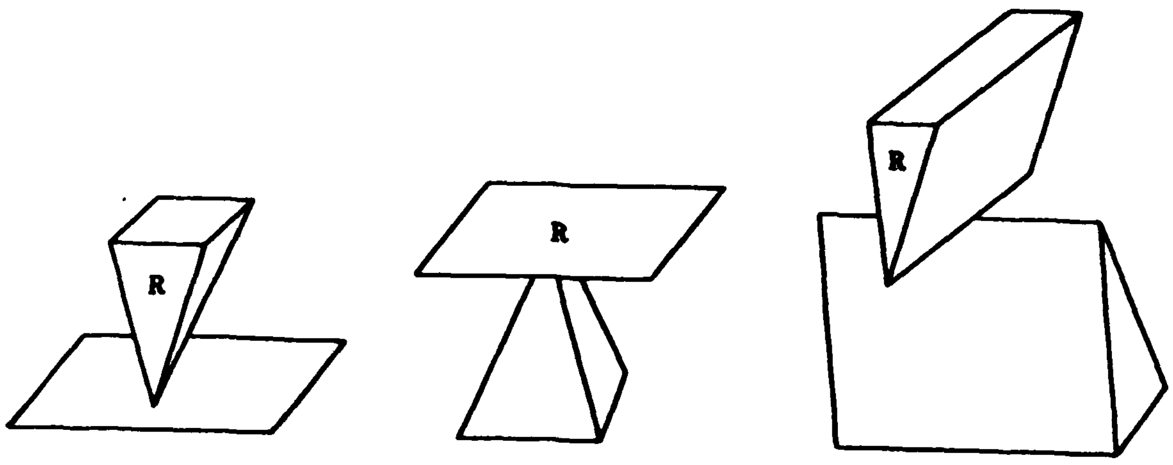


Figure 2.2: The combinations of a robot R and an obstacle that generate a face of contact space in three dimensions.

that the environment is given in a three-dimensional Euclidean configuration space representation.

2.3 Friction

Compliant motions cause the robot to contact obstacle surfaces. When a robot makes contact with an obstacle, the obstacle exerts a reaction force on the robot, consisting of a normal force, and a tangential force which is caused by friction between the robot and the obstacle. The vector sum of the robot force and the reaction force yields a resultant force which determines the direction of motion. Thus, a motion planner needs a geometric model of friction in order to reason about compliant motions.

Consider a one-point contact between the robot and an obstacle. This one-point contact defines a contact space face, with an associated normal vector (Figure 2.2). The reaction force that the obstacle exerts on the robot is governed by Coulomb's Law [see Baumeister 1978]. Coulomb's Law essentially states that:

1. The normal component of the reaction force with respect to the contact space face is equal and opposite to the normal component of the robot force.

2. The tangential component of the reaction force is no greater than μ times the normal component of the robot force, where μ is the coefficient of friction between the materials in contact. [1]

3. Within the limit imposed by (2), the obstacle will exert an equal and opposite tangential reaction force.

[1]For simplicity, we will assume that the coefficients of static and dynamic friction are equal.

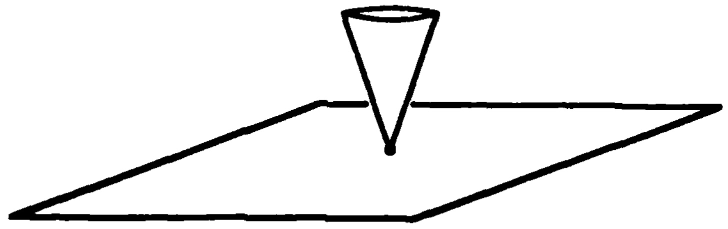


Figure 2.3: A section of the friction cone of a point on a contact space face.

The coefficient of friction μ has been measured experimentally for many combinations of materials. For example, see Baumeister [1978]. For common ungreased materials, μ ranges from .04 (teflon on teflon) to 1.10 (nickel on nickel). Baumeister lists μ as .73 for soft steel on soft steel, and .78 for hard steel on hard steel. The friction tables in the literature do not always agree. There are several factors which could account for this variance:

- The hardness of materials may vary.
- The degree of machining of contact surfaces may vary.
- There may be small amounts of grease on contact surfaces.

In practice, one should experimentally determine the coefficient of friction.

For a given contact between the robot and an obstacle, it is useful for a compliant motion planner to represent the set of all possible reaction forces. If the robot force is equal to the inverse of a reaction force in this set, then the robot will stick on the obstacle, because Coulomb's Law says that the obstacle will meet the robot force with an equal and opposite reaction force. If the robot force is not equal to the inverse of a reaction force in this set, then Coulomb's Law implies that the robot will either slide or break contact. These properties will be used in subsequent chapters to plan compliant motions.

For a one-point contact, Coulomb's Law implies that all possible reaction forces are contained in an infinite *friction cone* whose cone angle is equal to $\arctan(\mu)$. If $\mu = 0$, then the contact is frictionless. This means that the obstacle is only capable of exerting a reaction force which is normal to the contact surface. If $\mu = \infty$, then the obstacle is capable of meeting any robot force that is directed into the contact surface with an equal and opposite reaction force. The fact that a friction cone is infinite implies that a surface is capable of exerting a reaction force of any magnitude.

In order to plan compliant motions in configuration space, we need to assign friction cones to points in contact space. Each point on a contact space face inherits the friction cone of the one-point contact that generates the face. Figure 2.3 shows the friction cone of a point on a contact space face.

An edge in contact space is derived from contact between the robot and two obstacles. The reaction force of such a contact is equal to the vector sum of the

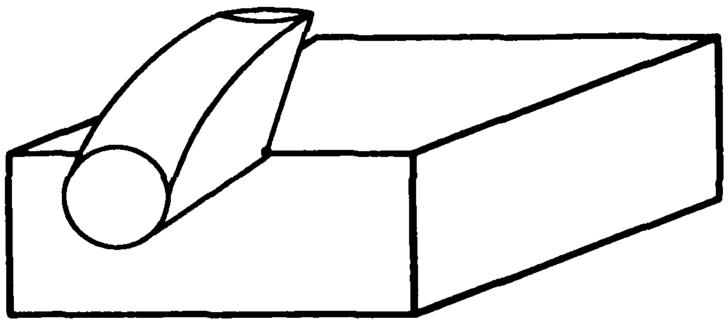


Figure 2.4: A section of the friction cone of a point on a convex edge in contact space. The cone is equal to the vector sum of the friction cones of the edge's containing faces.

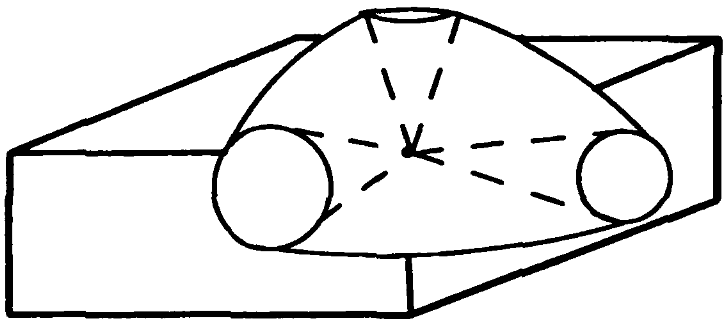


Figure 2.5: A section of the friction cone of a convex vertex in contact space. The cone is equal to the vector sum of the friction cones of the vertex's containing faces.

reaction forces of the two obstacles. Thus, the friction cone of an edge in contact space is equal to the vector sum of the friction cones of the contact space faces which contain the edge. Figure 2.4 shows the friction cone of a point on a convex edge in contact space. Note that the friction cone is not symmetric about an axis. Technically, it is not a right circular cone, but to maintain historical consistency we will continue to refer to it as such. It is similar to a right circular cone in that it consists of a set of rays which have a common origin, and is bounded by a smooth surface.

A vertex in contact space is derived from contact between the robot and three or more obstacles. The reaction force of such a contact is equal to the vector sum of the reaction forces of the obstacles. Thus, the friction cone of a vertex in contact space is equal to the vector sum of the friction cones of the contact space faces which contain the vertex. Figure 2.5 shows the friction cone of a convex vertex in contact space. Again, the friction cone is not symmetric about an axis.

2.4 Summary

In this chapter, we have described a model for three-dimensional geometric objects, such as the robot, workpieces, feeders, and fixtures. The model that we have chosen, the polyhedron, is useful because it simplifies the representation of an object. For example, when one needs to reason about compliant motions, each point on a particular polyhedron face, edge, or vertex is capable of the same set of reaction forces. We transform a polyhedral environment to the configuration space representation, by reducing the robot to a point, and growing obstacles accordingly. This representation makes it much simpler to reason about geometric constraints on the motion of the robot.

The possible reaction forces of a configuration space face, edge, or vertex are represented by a three-dimensional friction cone, which in the case of an edge or vertex is not actually a right circular cone, but has similar properties. The friction cone will allow us to reason about the motion of the robot when it is in contact.

Chapter 3

Modeling Robot Motions

In this chapter, we will search for an abstract model for specifying robot motions. The model is to be used by both human programmers and automatic motion planners. We will be looking for an abstraction which meets the following requirements:

1. Instances of the model should be easy to specify.

2. Using the model, one should be able to specify robot motions with wide utility.

3. The model should be implementable on a robot, with bounded error.

We will begin in Section 3.1 by reviewing a number of currently existing models. From this list, we will focus in on two *particular* models which fit our needs in planning compliant motions. We will then examine the properties of these models in detail, in Sections 3.2 and 3.3. In Section 3.4, we will discuss the implementation of these models on a robot.

3.1 Review of Current Models

A robot motion can be modeled by two components, a desired trajectory, and a set of termination conditions. A desired trajectory is modeled by a differential equation which describes the configuration of the robot with respect to time. In Section 3.1.1, we will review currently existing trajectory models, and pick two that are suitable for our purposes.

The termination conditions of a commanded motion specify situations in which the robot is to terminate its trajectory. In Section 3.1.2, we will discuss termination conditions in some generality, and identify the types of termination conditions that we will be using.

Our models will allow for bounded errors in the execution of a commanded motion. There are two types of errors that can cause a commanded motion to stray from its commanded trajectory, control errors and sensing errors. A control error is an inability of the robot to achieve a commanded position or velocity. We will assume that the following bounds hold on control errors:

ϵ_p: The maximum distance between a commanded position and the actual position attained by the control system in free space.

θ_v: The maximum angle between a commanded velocity and the actual velocity attained by the control system in free space. We will assume that $\theta_v \leq \frac{\pi}{2}$.

We will assume that the following bounds hold on sensing errors:

ϵ_s: The maximum distance between a sensed position and an actual position.

θ_f: The maximum angle between a sensed reaction force vector and an actual reaction force vector.

The notations ϵ_s, θ_f, ϵ_p, and θ_v will be used throughout the thesis to denote the error bounds described here.

3.1.1 Trajectory Models

The best known trajectory model is Newton's second law of motion,

$$\mathbf{f} = M\ddot{\mathbf{x}}, \tag{3.1}$$

where $\mathbf{f}$ is the total force on the robot, $\mathbf{x}$ is its configuration, and M is an inertial matrix, which depends on $\mathbf{x}$. Since Newton's law is second order, the trajectory of the robot in response to an applied force is parabolic. Parabolic trajectories are difficult to specify, however. Consider Figure 1.1, for example. How would you specify the parameters of a parabolic trajectory for this insertion? It would be much easier to command a linear trajectory. To do this, we need to reduce the order of the differential equation to 1 or 0:

$$\mathbf{f} = B\dot{\mathbf{x}} \tag{3.2}$$

$$\mathbf{f} = K\mathbf{x} \tag{3.3}$$

B and K are called *damping* and *stiffness* matrices, respectively.

Equations 3.2 and 3.3 do not match the physical world as accurately as Equation 3.1. Robot trajectory models are similar to Newton's law in that they specify a dynamic behavior, but different in that they only expect the behavior to hold to within some tolerance. Another difference is that Newton expected the physical world to uphold his law, while a robot programmer expects a robot control system to implement a robot trajectory model.

Four currently existing types of trajectory models are *phase trajectory models*, *force trajectory models*, *hybrid trajectory models*, and *impedance trajectory models*. Phase trajectory models include the *position trajectory model*,

$$\mathbf{x} = \mathbf{x}_c, \tag{3.4}$$

the *velocity trajectory model*,

$$\dot{\mathbf{x}} = \dot{\mathbf{x}}_c, \tag{3.5}$$

and the *acceleration trajectory model*,

$$\ddot{\mathbf{x}} = \ddot{\mathbf{x}}_c. \tag{3.6}$$

Equation 3.4 says that the actual position $\mathbf{x}$ is to be equal to a commanded position $\mathbf{x}_c$. Equation 3.5 says that the actual velocity $\dot{\mathbf{x}}$ is to be equal to a commanded velocity $\dot{\mathbf{x}}_c$. Equation 3.6 says that the actual acceleration $\ddot{\mathbf{x}}$ is to be equal to a commanded acceleration $\ddot{\mathbf{x}}_c$. Unfortunately, these models do not fully specify the behavior of the robot. A commanded position may be unattainable because an obstacle is in the way. A commanded velocity may be modified significantly by the reaction force of an obstacle. The trajectory models do not tell us what happens in these cases. Essentially, the configuration of the robot is underdetermined when the robot is in contact under these trajectory models. Additional dynamic constraints are needed. One possibility would be to assume Newtonian dynamics (Equation 3.1). Alternatively, we could assume dynamics of the same order as the commanded phase variable.

Force trajectory models specify the external force on the robot. A force trajectory is given by the equation

$$\mathbf{f}_r = \mathbf{f}_c. \tag{3.7}$$

This says that the actual external force $\mathbf{f}_r$ on the robot is to be equal to a commanded force $\mathbf{f}_c$. A force trajectory also fails to fully specify the behavior of the robot. Equation 3.7 does not specify the position of the robot, nor any of its derivatives. In free space, where there is no external force on the robot, a force trajectory says nothing. Again, additional dynamic constraints are needed.

Force trajectories have been combined with phase trajectories in the *hybrid trajectory model* [Mason 1979, Raibert and Craig 1981], in which certain robot axes are assigned force trajectories, and others are assigned phase trajectories. It is as if we have several coordinated one degree-of-freedom robots, some in free space, and some in contact space. The problem with hybrid trajectories is that if a robot axis in a phase trajectory establishes contact, the motion cannot continue smoothly into contact space. Instead, the robot must terminate the motion, and initiate a new force trajectory along the axis. This results in jerky robot motion, and leads to larger and more complicated motion strategies.

The reason why hybrid trajectories do not allow smooth continuation from free space to contact space is that they do not specify an explicit relationship between the position of the robot and the reaction force on it. Trajectory models which do this are called *impedance trajectory models*. They include the *generalized spring* trajectory model [Salisbury 1980],

$$\mathbf{f}_r = K(\mathbf{x} - \mathbf{x}_c), \tag{3.8}$$

the *generalized damper* trajectory model [Whitney 1977],

$$\mathbf{f}_r = B(\dot{\mathbf{x}} - \dot{\mathbf{x}}_c), \tag{3.9}$$

and the *generalized mass* trajectory model,

$$\mathbf{f}_r = M(\ddot{\mathbf{x}} - \ddot{\mathbf{x}}_c), \tag{3.10}$$

where $\mathbf{f}_r$ is the reaction force imposed by the environment on the robot; $\mathbf{x}_c$, $\dot{\mathbf{x}}_c$, and $\ddot{\mathbf{x}}_c$ are the commanded phase variables; and $\mathbf{x}$, $\dot{\mathbf{x}}$, and $\ddot{\mathbf{x}}$ are the actual phase variables. K, B, and M are spring, damping, and inertial matrices, respectively.

Impedance trajectory models pick an order for the differential equation of the robot, and model the robot as an impedance of the highest order. For example, the generalized mass trajectory model defines a second order system with the robot modeled as a mass. Each of these trajectory models can be seen as an interpretation of a phase trajectory, further constrained to obey a differential equation of the same order as the commanded phase variable. For example, the generalized mass trajectory model is an interpretation of an acceleration trajectory, further constrained to obey Newtonian dynamics. To show this, we derive Equation 3.10 from Equation 3.1:

$$\begin{aligned} \mathbf{f} &= M\ddot{\mathbf{x}} \\ \mathbf{f}_r + M\ddot{\mathbf{x}}_c &= M\ddot{\mathbf{x}} \\ \mathbf{f}_r &= M(\ddot{\mathbf{x}} - \ddot{\mathbf{x}}_c) \end{aligned}$$

The other two impedance trajectory models can be derived similarly, by starting with a lower order differential equation.

Hogan [1984] has combined all of these impedance models into a single model, in which the robot is modeled as a system consisting of a spring, damper, and inertia. His trajectory equation is

$$\mathbf{f}_r = M\ddot{\mathbf{x}} + B(\dot{\mathbf{x}} - \dot{\mathbf{x}}_c) + K(\mathbf{x} - \mathbf{x}_c). \tag{3.11}$$

For the time being, we will focus on the individual impedance models. However, in Section 3.3, we will show that the first order analog to this equation is needed to properly model a generalized spring.

Under an impedance trajectory model, the user can specify both the commanded phase variable, and the stiffness, damping, or inertial matrix. It is useful to specify the stiffness, damping, or inertial matrix as a positive diagonal matrix, so that there are no cross-coupling effects between commanded axes. It is also useful to assume that the diagonal elements of the matrices are identical. K can then be treated as a positive constant k, B as a positive constant b, and M as a positive constant m. Under this convention, it is the commanded phase variable alone which controls whether the robot will stick or slide on an environment surface. This is discussed in detail below for each impedance trajectory model. The stiffness, damping, or inertial constant controls the magnitude of the environmental reaction force, which affects the stability of the system. The value of the constant does not affect the robot trajectory, as long as stability is maintained.

The generalized mass trajectory model is second order, and for reasons described above, is not convenient to use. We have chosen to use the generalized damper and spring trajectory models in our programming system. The user may command motions from either model, as desired. Sections 3.2 and 3.3 discuss the properties of these trajectory models in more detail.

3.1.2 Termination Conditions

The termination conditions of a commanded motion specify situations in which the robot is to terminate its trajectory. They are a generalization of what Will and Grossman [1975] called *guarded moves*.

To specify a termination condition, our programming system allows the following *termination parameters* to be specified in a motion command:

P_s is a set of positions.

F_s is a set of reaction force directions.

The basic idea is that if the robot senses a position that is in the set P_s, and a reaction force vector that is in the set F_s, then it is to terminate the motion.

A motion planner faces the question of how these parameters can be used to terminate a motion in a goal region G. Mason [1983] and Erdmann [1984] addressed this question in some generality. We will review some of their findings here.

The termination parameters should contain positions and reaction forces of goal points whose interpretations are all in G. even under maximal sensing error. This way, the robot will not stop in a nongoal configuration by accident. There is information available that supplements the sensed position and reaction force. For one thing, the planner presumably has knowledge of a start region S, which contains the initial configuration of the robot. S is a calculated bound, based on previous motions in the planned strategy, under sensing and control uncertainty.

The planner also knows the commanded motion m, which is either a position or a velocity. Define $F_m(S)$ to be the set of contact configurations that are reachable from S under m. $F_m(S)$ is called the *forward projection of S under m*. Chapter 4 presents an algorithm for computing a forward projection, given the task geometry. The planner can use the forward projection to compute larger sets of termination parameters. These sets contain positions and reaction forces of goal points whose interpretations are all in G or outside of $F_m(S)$.

At runtime, for each motion, the robot knows the actual initial configuration $\mathbf{s}$, to within the position sensing uncertainty ϵ_s. Let $B(\mathbf{s})$ be a ball of radius ϵ_s, centered at $\mathbf{s}$. If we use the forward projection

$$F_m(B(\mathbf{s}) \bigcap S) \tag{3.12}$$

rather than $F_m(S)$ to generate the termination parameters, then the forward projection will be smaller, and the termination parameters will be larger. A termination condition which uses the actual initial configuration s is called a *termination condition with history*. However, this is a difficult termination condition to plan with, since s is not available until runtime. Also, forward projections cannot currently be computed in real time. Our programming system will not use s in termination conditions.

Another source of information that our programming system will ignore is elapsed time. Erdmann presented an example in which a robot used elapsed time to terminate in a goal region. The example could not have been performed without timing information. However, it remains an open problem to design a planner that can utilize it.

3.2 The Generalized Damper Trajectory Model

In this section, we will discuss the properties of the generalized damper trajectory model. We will answer the following questions:

- What free space trajectories are possible under the trajectory model?
- What contact space trajectories are possible under the trajectory model?
- In contact space, when does the robot stick and when does it slide under the trajectory model?

Much of what follows is due to Lozano-Pérez, Mason, and Taylor [1983]. For simplicity, we will assume that commanded velocities are limited to unit velocity vectors. This does not limit the tasks that we can accomplish; it merely limits the speed at which we can accomplish them. With this limitation, we can focus on the effect of the direction of a commanded velocity.

3.2.1 Free Space Trajectories

In the absence of control error, a robot trajectory through free space under the generalized damper trajectory model is given by a ray originating at the initial configuration and parallel to the commanded velocity $\dot{\mathbf{x}}_c$. With control error, there may be directional errors in achieving $\dot{\mathbf{x}}_c$. The actual velocities that are possible in free space are contained in a cone, as shown in Figure 3.1. This cone is referred to as the *velocity cone*. The angle θ_v of the velocity cone provides a parametric bound on velocity error. The knowledge of this cone would allow a planner to synthesize collision-free motions if desired.

We will assume that the robot can instantaneously switch between velocities from its velocity cone. This is a conservative assumption, since in practice a delay would occur on a switch between two velocities. However, second order dynamics would be needed to model such a delay, and we would like to use first order dynamics. An alternative would be to require that a velocity remain constant along a free space trajectory, but this would be a bad assumption for general-purpose robots.

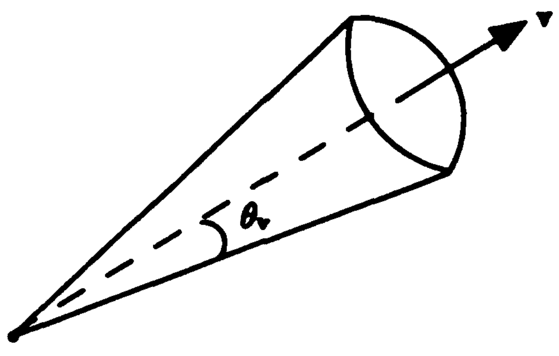


Figure 3.1: All free space trajectories from an initial configuration **s** under the generalized damper trajectory model are contained in a cone of angle θ_v.

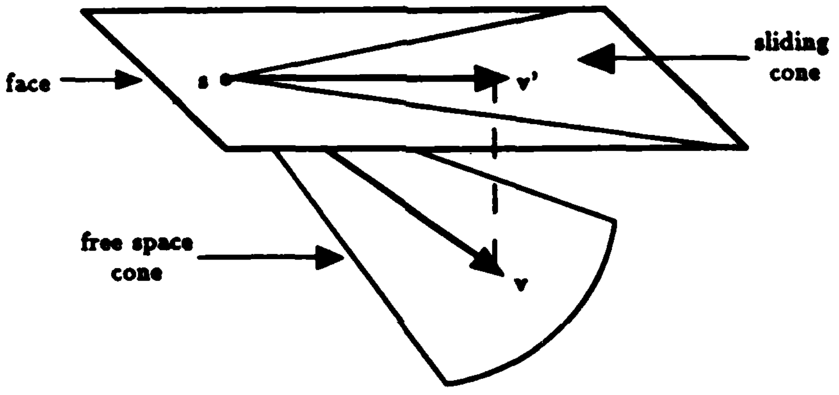


Figure 3.2: The initial configuration of the robot is **s**. The commanded velocity is **v**. Under the generalized damper trajectory model, all sliding trajectories are contained in the projection of the velocity cone onto the surface. The axis of the sliding cone is the projection **v**′ of **v** onto the surface.

3.2.2 Contact Space Trajectories

Assume that the robot is in contact with a face, and that a commanded velocity is directed into the plane of the face. For now, assume that the surface is frictionless. (We will discuss the effects of friction below.) The sliding trajectories that are possible under the generalized damper trajectory model are contained in the projection of the velocity cone onto the sliding surface, as shown in Figure 3.2.

3.2.3 Sticking and Sliding

Assume initially that there is no control error. In contact space, the robot either sticks at a surface point or slides away from it, along the projection of the commanded velocity onto the surface. Whether the robot sticks or slides depends upon the direction of $\dot{\mathbf{x}}_c$. The damper force imparted by the commanded velocity, $b\dot{\mathbf{x}}_c$,

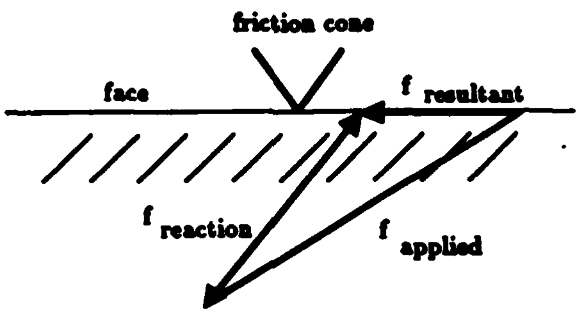


Figure 3.3: A cross section of a face, showing the resultant force when the commanded velocity is outside of the negative friction cone.

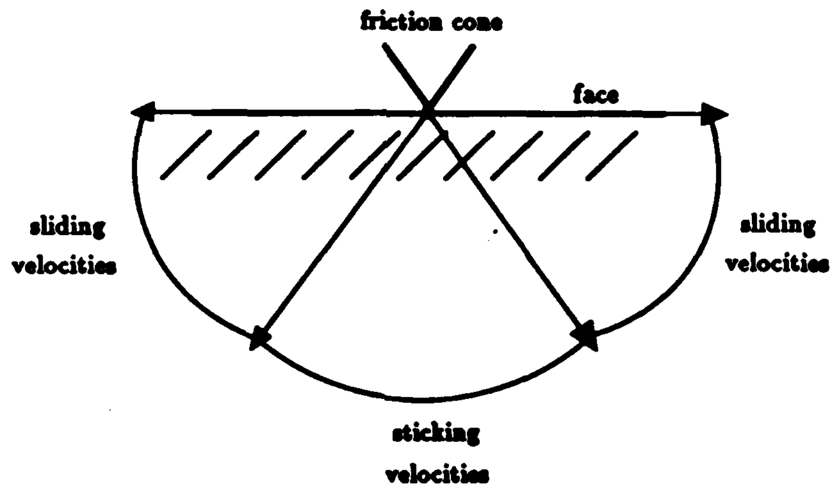


Figure 3.4: A cross section of a face, showing commanded velocities which cause sticking and sliding on the face, under the generalized damper trajectory model.

is proportional to $\dot{\mathbf{x}}_c$. If $\dot{\mathbf{x}}_c$ is directed into the negative friction cone of a surface point, then the surface point generates an equal and opposite reaction force, and the robot sticks. If $\dot{\mathbf{x}}_c$ is directed outside of the negative friction cone, then the reaction force is equal to the negative of the projection of the robot force onto the friction cone, as shown in Figure 3.3. The resultant force is along the surface, as shown in the figure. The resultant velocity is proportional to the resultant force, causing sliding along the surface.

The knowledge of this behavior allows a planner to pick $\dot{\mathbf{x}}_c$ based on the desire to either stick or slide on a particular surface. For example, Figure 3.4 shows commanded velocities for which sticking and sliding will occur on a face. If $\dot{\mathbf{x}}_c$ is outside of the negative friction cone, then the robot will slide along the length of a face until it reaches another face, or a termination condition becomes satisfied. This allows the robot to use a surface as a guide, as illustrated in Figure 1.5.

Now, let us consider the effect of control error. In order to ensure sticking or

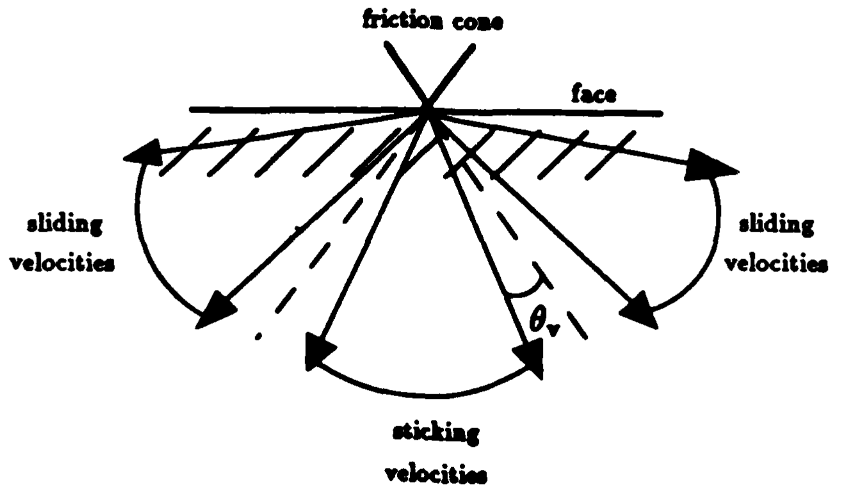


Figure 3.5: A cross section of a face, showing commanded velocities which ensure sticking and sliding under the generalized damper trajectory model with bounded velocity error.

---

sliding under bounded velocity error, one must ensure that $\dot{\mathbf{x}}_c$ is directed into or outside of the negative friction cone, even if a maximal velocity error of θ_v occurs. For example, Figure 3.5 shows velocities which ensure sticking and sliding on a face despite velocity error.

Figure 3.6 shows a case where generalized damper commands can be used to reach a goal, despite significant velocity errors.

3.3 The Generalized Spring Trajectory Model

In this section, we will discuss the properties of the generalized spring trajectory model. We will answer the same questions that we answered for the generalized damper trajectory model.

The zero order dynamic system given by Equation 3.8 fails to fully define the behavior of the robot. First, it does not define a desired behavioı in free space for the case where $\mathbf{x}$ is not equal to $\mathbf{x}_c$, for there are no reaction forces in free space, and the equation is unsatisfied. Also, it fails to define a desired behavior in certain contact configurations. For example, in Figure 3.7, $\mathbf{x}_c - \mathbf{x}$ is outside of the negative friction cone. According to Equation 3.8, $\mathbf{f}_r$ should be proportional to $\mathbf{x} - \mathbf{x}_c$, but this contradicts Coulomb's Law.

What kind of behavior would we like of our trajectory model? In free space, if $\mathbf{x}$ is not equal to $\mathbf{x}_c$, we would like the robot to move in a straight line from $\mathbf{x}$ to $\mathbf{x}_c$. In contact space, if $\mathbf{x}_c - \mathbf{x}$ is outside of the negative friction cone, we would like the robot to slide towards a contact configuration in which $\mathbf{x}_c - \mathbf{x}$ is directed into the negative friction cone, and stick there.

The main problem with Equation 3.8 is that it is zero order, and thus does

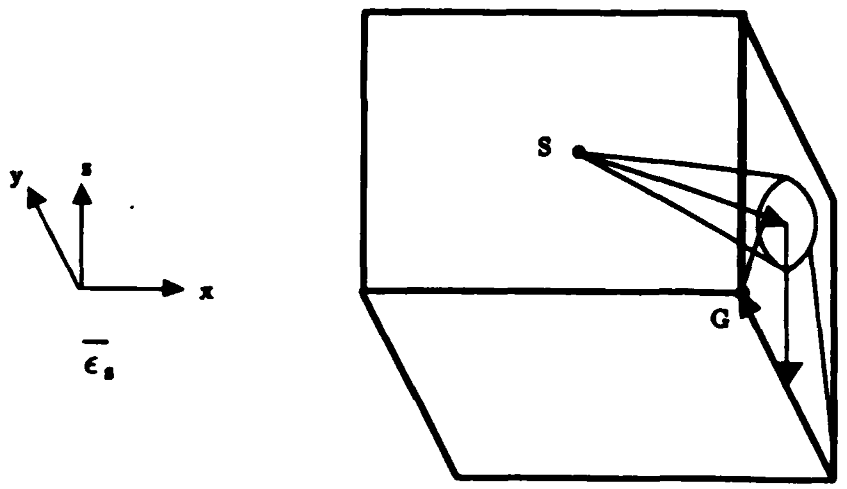


Figure 3.6: This is the same example as Figure 1.5. The goal is the concave corner. The robot begins in free space. Using generalized damper motions, the robot is able to proceed to the corner despite velocity errors. The trajectory of the first motion is bounded by a three-dimensional cone. The trajectory of the second motion is bounded by a two-dimensional cone. The third motion is one-dimensional, and thus has no velocity error.

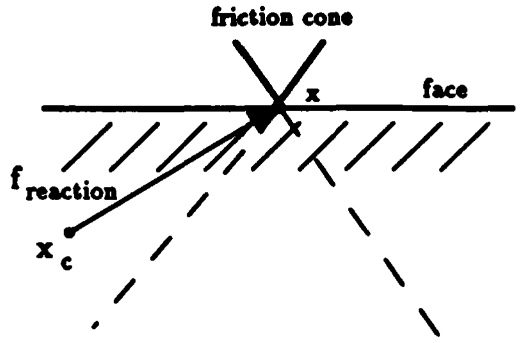


Figure 3.7: The cross section of a face. The commanded position is $\mathbf{x}_c$. The robot is in configuration $\mathbf{x}$, and imparts a spring force towards $\mathbf{x}_c$. Under Equation 3.8, the reaction force of the face should be $\mathbf{f}_{reaction}$. But this is impossible, for $\mathbf{f}_{reaction}$ is outside of the friction cone.

not describe any change in configuration. In short, the equation does not describe motion. We need to formulate an equation of higher order. We do not want an equation of second order, because we do not want to deal with parabolic motion. Thus, we choose a first order equation, one in which the commanded phase variable is position rather than velocity. To do this, we model the robot as a damped spring. One simple way to formulate the dynamic equation is to start with the dynamic equation of a generalized damper (Equation 3.9), and substitute the quantity $\mathbf{x}_c - \mathbf{x}$ for $\dot{\mathbf{x}}_c$. This has the effect of pulling the robot toward $\mathbf{x}_c$ instead of letting it fan out into the velocity cone. The resulting dynamic equation is

$$\mathbf{f}_r = b(\dot{\mathbf{x}} - \mathbf{x}_c + \mathbf{x}). \tag{3.13}$$

Another way to derive this dynamic equation is to start with the first-order equivalent of Newton's law (Equation 3.2), model the robot as a spring with stiffness k, and solve the force-balance equation for the reaction force:

$$\begin{aligned} \mathbf{f} &= b\dot{\mathbf{x}} \\ \mathbf{f}_r + k(\mathbf{x}_c - \mathbf{x}) &= b\dot{\mathbf{x}} \\ \mathbf{f}_r &= k(\mathbf{x} - \mathbf{x}_c) + b\dot{\mathbf{x}} \end{aligned} \tag{3.14}$$

Note that Equation 3.14 is the first-order analog to Hogan's impedance equation (Equation 3.11). The imparted spring force of the robot is equal to $k(\mathbf{x}_c - \mathbf{x})$. Since we are free to choose k and b, we can set $k = b$, thus obtaining Equation 3.13.

In free space, the reaction force is 0, and Equation 3.13 reduces to

$$\dot{\mathbf{x}} = \mathbf{x}_c - \mathbf{x}. \tag{3.15}$$

This says that the velocity of the robot is directed along a vector from the actual position to the commanded position, as desired.

In contact space, if $\mathbf{x}_c - \mathbf{x}$ is into the negative friction cone, then the surface is able to respond to the spring force $b(\mathbf{x}_c - \mathbf{x})$ with the equal and opposite reaction force $b(\mathbf{x} - \mathbf{x}_c)$. Equation 3.13 reduces to

$$\dot{\mathbf{x}} = 0, \tag{3.16}$$

which means that the robot sticks. If $\mathbf{x}_c - \mathbf{x}$ is outside of the negative friction cone, then f_r is equal to the negative of the projection of the spring force $b(\mathbf{x}_c - \mathbf{x})$ onto the friction cone. This causes a resultant force along the surface, as shown in Figure 3.8. The first order dynamics turn this force into a sliding velocity along the surface. Equation 3.13 is thus the trajectory model that we were looking for.

3.3.1 Free Space Trajectories

What are the free space trajectories of a robot under Equation 3.13? Suppose that the robot is initially in configuration $\mathbf{s}$, and is commanded to go to configuration $\mathbf{p}$. Assume for now that there are velocity errors, but that positioning and position

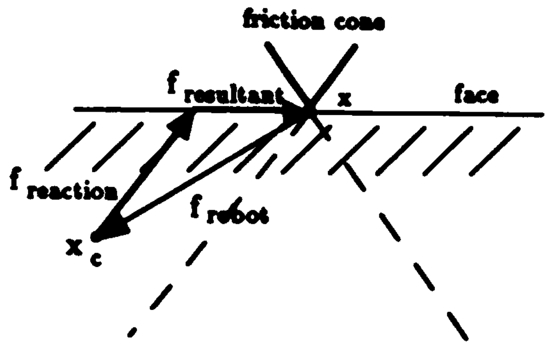


Figure 3.8: The cross section of a face. The commanded position is $\mathbf{x}_c$. The robot is in configuration $\mathbf{x}$, and imparts a spring force towards $\mathbf{x}_c$. The reaction force of the face is $\mathbf{f}_{reaction}$. The resultant force is along the surface of the face.

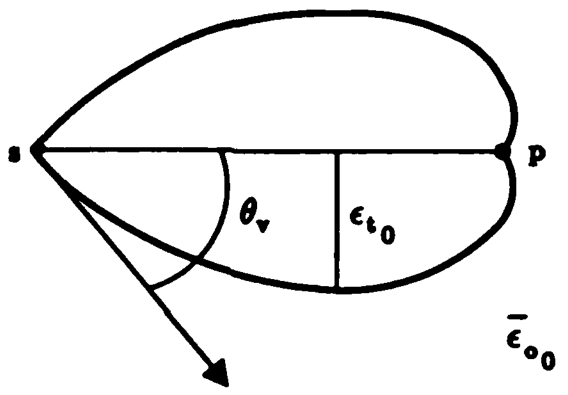


Figure 3.9: Under the generalized spring trajectory model with perfect positioning and position sensing, the free space trajectories of a robot starting in configuration **s** and commanded towards **p** are bounded the spiral curves shown here. The maximum trajectory error is ϵ_{t_0}, and the maximum overshoot is ϵ_{o_0}.

sensing is perfect. In its attempt to move directly from **s** to **p**, there may be a directional velocity error, bounded by θ_v. However, this does not mean that the trajectories fan out into a cone, like generalized damper trajectories. Under the generalized spring trajectory model, the robot is constantly being pulled towards **p**. This has a corrective effect on trajectories which attain maximal directional error. Figure 3.9 shows the possible trajectories that result from the corrective action. The trajectories are bounded by a collection of spiral curves.

Two parameters of a generalized spring trajectory that a planner needs to know are the maximum trajectory error ϵ_{t_0} and the maximum overshoot ϵ_{o_0}, as shown in Figure 3.9. Let p be the distance from **s** to **p**, and define:

$$t_t = \frac{\theta_v}{\tan \theta_v} \tag{3.17}$$

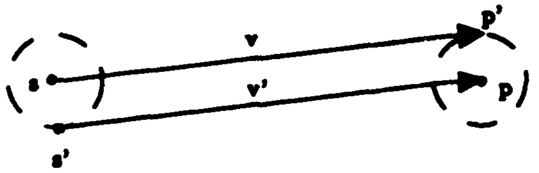


Figure 3.10: The initial position of the robot is **s**, with the indicated position sensing uncertainty. The commanded position is **p**. The robot thinks that it is at position **s**$'$, and commands velocity **v**$'$. In actuality, the robot is at **s**, and velocity **v** results. This is just as if the robot were aiming for the errant commanded position **p**$'$.

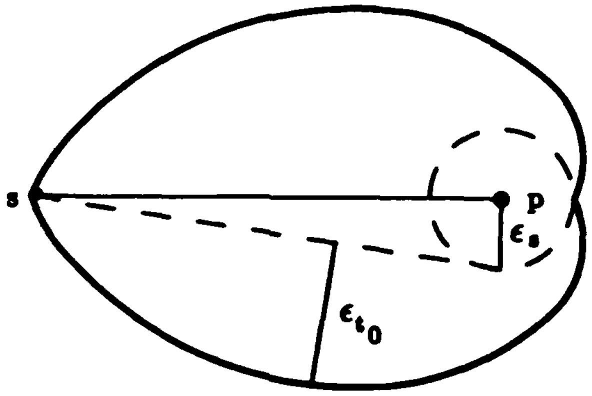


Figure 3.11: Generalized spring free space trajectories with bounded velocity and position sensing error.

$$t_o = \frac{\theta_v + \frac{\pi}{2}}{\tan\theta_v} \tag{3.18}$$

Then the spring trajectory parameters are given by:

$$\epsilon_{t_0} = pe^{-t_t}\sin\theta_v \tag{3.19}$$

$$\epsilon_{\infty_0} = p(1 + e^{-t_o}\sin\theta_v) \tag{3.20}$$

These expressions are derived in Appendix B.

Let us now consider the effect of bounded position sensing error. The position sensors of the robot are only accurate to within ϵ_s. This means that the robot may misjudge its current configuration by as much as ϵ_s. This is equivalent to misjudging the commanded position **p** by ϵ_s. (See Figure 3.10.) The resulting free space trajectories are shown in Figure 3.11.

A generalized spring motion is in a sense more accurate than a generalized damper motion because it does not fan out into a full cone. However, the accuracy of a generalized spring motion depends on the distance p to the commanded position.

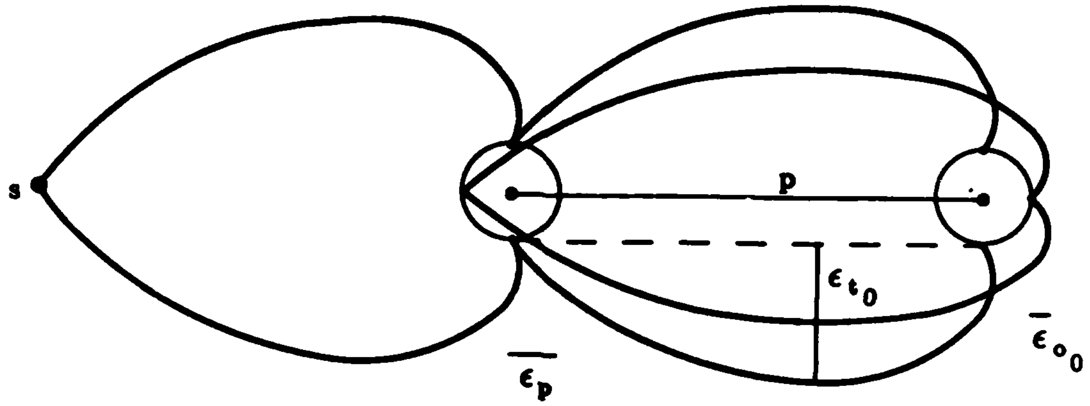


Figure 3.12: Some extreme trajectories from an interpolated generalized spring motion through free space. The initial configuration of the trajectories is **s**. p is the interpolation distance.

This makes it difficult for a planner to utilize the increased accuracy. Fortunately, a control system can remove this dependency by interpolating additional commanded positions. A commanded motion from **s** to **p** can be broken up into segments of fixed length p by a trajectory planner. The endpoints of the segments can then be passed to the position controller in succession, so long as the robot remains in free space (i.e. does not sense a reaction force).

What is the maximum trajectory error and overshoot for an interpolated free space trajectory under sensing and control uncertainty? To answer this question, recall that the final positioning accuracy of the robot is ϵ_p. This means that interpolated positions can only be achieved to within ϵ_p. Since position sensing uncertainty contributes to positioning error, it is assumed that $\epsilon_s \leq \epsilon_p$. Let us for a moment make the conservative assumption that $\epsilon_s = \epsilon_p$. Then we can sketch out some extreme trajectories, as shown in Figure 3.12. It can be seen from the figure that the maximum trajectory error ϵ_t is bounded by

$$\epsilon_t = \epsilon_p + pe^{-t_t} \sin\theta_v, \tag{3.21}$$

and the maximum overshoot ϵ_o is given by

$$\epsilon_o = \epsilon_p + (p + 2\epsilon_p)(1 + e^{-t_o} \sin\theta_v). \tag{3.22}$$

If the trajectory planner is implemented as an analog circuit, then a continuous stream of control positions can be passed to the position controller, reducing ϵ_t to ϵ_p.

For planning purposes, it suffices to bound free space trajectories by a cylinder of radius ϵ_t, as shown in Figure 3.13. ϵ_t is given by Equation 3.21. The knowledge of this cylinder would allow a planner to synthesize collision-free motions if desired.

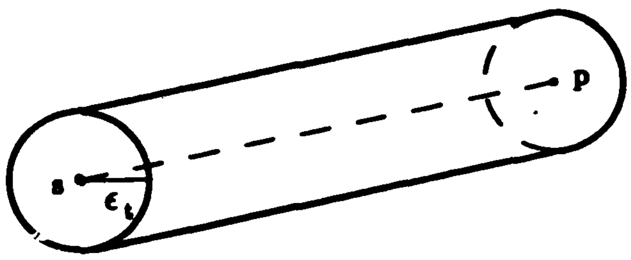


Figure 3.13: All possible free space trajectories from an initial configuration s under the generalized spring trajectory model are contained in a cylinder of radius ϵ_t.

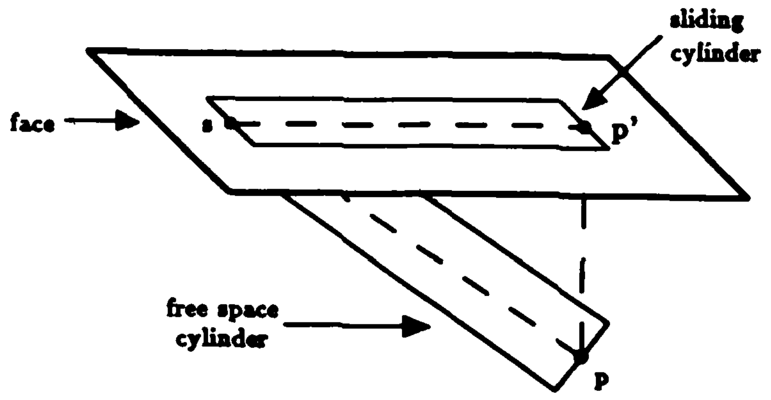


Figure 3.14: The initial configuration of the robot is s. The commanded position is **p**. Under the generalized spring trajectory model, all sliding trajectories are contained in the projection of the freespace trajectory cylinder onto the surface. **p**$'$ is the projection of the commanded position onto the surface.

3.3.2 Contact Space Trajectories

When the robot strikes a surface, the cylinder of possible free space trajectories is projected onto the surface, forming a planer cylinder of radius ϵ_t, as shown in Figure 3.14. When the robot breaks contact, the trajectory controller must replan the trajectory, using the breakaway configuration as the initial point of the new trajectory. This is necessary because the interpolation distance may exceed p when free space is reentered. If the robot is not able to detect the transition, then the trajectory error for the next interpolation point may exceed ϵ_t.

3.3.3 Sticking and Sliding

The trajectory controller *never explicitly checks for sticking termination*. If a trajectory plan has been issued to completion, and the robot has stuck somewhere, then that is where it ends up. Since plans are always issued to completion, the final commanded position **p** is what ultimately determines where the robot might stick along a trajectory. A planner can thus compute commanded positions for sticking

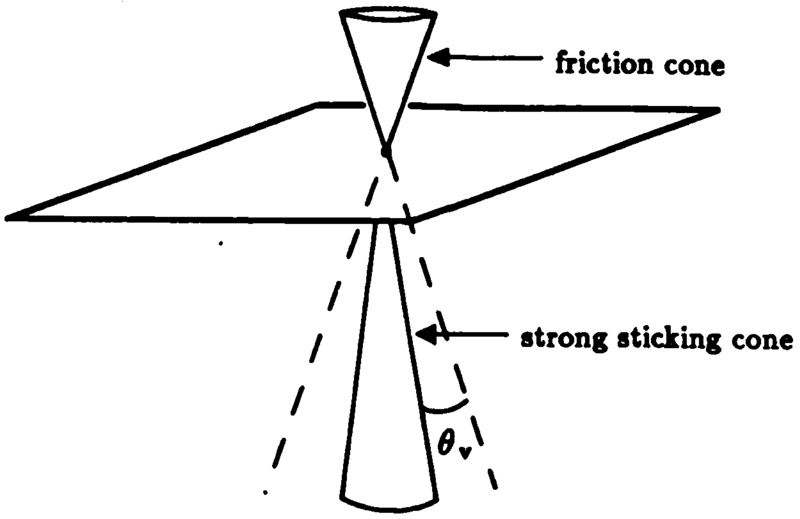


Figure 3.15: The cone of commanded positions which ensure sticking on a face despite bounded velocity error.

termination without worrying about the effects of interpolation.

Suppose that the robot is in a contact configuration $\mathbf{x}$ on a face, and that the commanded position is $\mathbf{p}$. In the absence of sensing and control error, the force imparted by the robot is equal to $b(\mathbf{p} - \mathbf{x})$, from Equation 3.13. If we negate the friction cone at $\mathbf{x}$, then we have the set of robot forces that will cause sticking at $\mathbf{x}$. Thus, the negative friction cone contains the set of commanded positions that will cause sticking at $\mathbf{x}$, in the absence of sensing and control error.

With bounded velocity error, the force imparted by the robot may err by an angle that may be as large as θ_v. To ensure sticking at $\mathbf{x}$, we need to decrease the angle of the negative friction cone by θ_v. The resulting cone, shown in Figure 3.15, contains commanded positions which are guaranteed to cause sticking, even under maximal velocity error.

Under bounded position sensing error, the robot may misjudge its position by as much as ϵ_s. This may cause the robot to aim for a commanded position which is off by as much as ϵ_s. This will alter the force imparted by the robot. We can take this into account by shrinking the cone of commanded positions by ϵ_s.

To ensure sliding away from $\mathbf{x}$, we must pick a $\mathbf{p}$ which satisfies two constraints:

1. $\mathbf{p}$ must not be in the negative friction cone of $\mathbf{x}$, expanded in angle by θ_v, and then grown volumetrically by ϵ_s. This ensures that $\mathbf{p}$ will not cause sticking, even under maximal velocity and position sensing error.

2. $\mathbf{p}$ must be interior to the face by at least ϵ_s. This ensures that the robot will not fly off of the face, even under maximal position sensing error.

We assumed above that the contact configuration of the robot was given. We then discussed the effects of different commanded positions on sticking and sliding

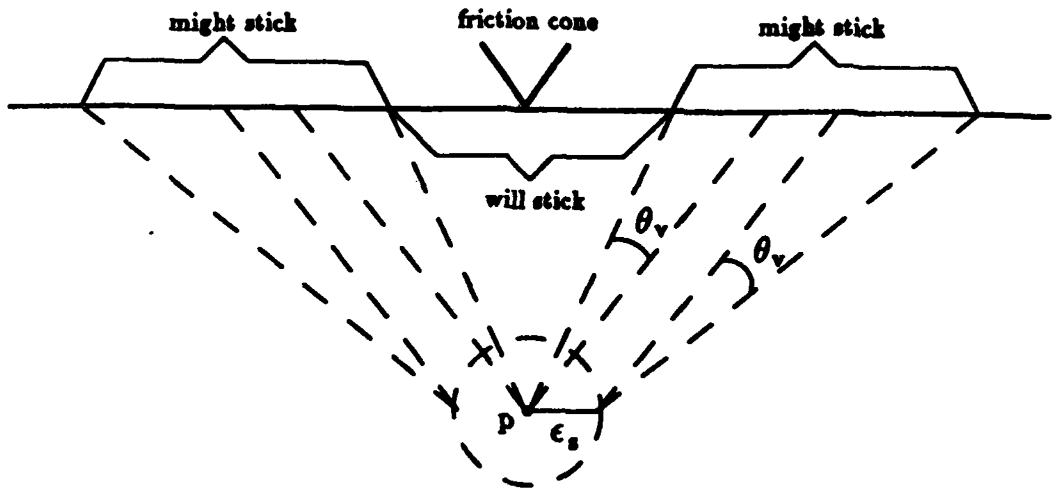


Figure 3.16: A cross section of a face, showing contact configurations at which sticking might and will occur under the generalized spring trajectory model, with bounded position sensing and velocity error. The commanded position is **p**. The strong sticking configurations are the configurations that will stick. All of the *sticking configurations are weak sticking configurations.*

behavior. Alternatively, it is sometimes useful to assume that a commanded position is given, and to ask which contact configurations will stick and which will slide. Figure 3.16 answers this question for the contact configurations on a face. The figure shows two types of sticking configurations, *strong sticking configurations*, at which sticking is guaranteed under the given commanded position, and *weak sticking configurations*, at which sticking is possible under the given commanded position.

3.4 Implementation

Stability issues have prevented the robust implementation of computer-controlled compliant motions to date. (See Whitney [1985] and An [1986].) As a result, we will not be able to demonstrate conclusively that our trajectory models are implementable. Instead, we will propose a design for the implementation of the trajectory models, and exhibit an experimental implementation of the design.

Let us review the main ideas behind the generalized damper and spring trajectory models. It is assumed that the robot can move in a commanded direction, to within an angular tolerance. Since we are assuming a first order dynamic system, the robot imparts a force in the direction of motion. The generalized damper trajectory model commands a constant motion direction. The generalized spring trajectory model modifies the commanded motion direction such that it always points at the commanded position. In contact, Coulomb's law should tell us whether the commanded direction will result in sticking or sliding.

There are thus two main tasks for the control system:

- In free space, the control system must cause the robot to move in a commanded direction, to within a specified angular tolerance.
- In contact, the control system must cause the robot to impart a force in a commanded direction, to within the same specified angular tolerance.

Suppose that we have a velocity controller at our disposal, such as that of Whitney [1969]. Were it not for the second task, we could merely command the velocity controller to move in the commanded direction. However, since the real world is second order, this velocity would not necessarily cause a force in the commanded direction.

An alternative scheme is to command the velocity controller to follow the desired *actual trajectory*, based on force feedback. A commanded damper velocity $\dot{\mathbf{x}}_c$ can be implemented by sending the desired actual velocity

$$\dot{\mathbf{x}} = \dot{\mathbf{x}}_c + \frac{\mathbf{f}_r}{b} \tag{3.23}$$

to the velocity controller, where $\mathbf{f}_r$ is the sensed reaction force vector, and b is the damping constant. A commanded spring position $\mathbf{x}_c$ can be implemented by sending the desired actual velocity

$$\dot{\mathbf{x}} = \mathbf{x}_c - \mathbf{x} + \frac{\mathbf{f}_r}{b} \tag{3.24}$$

to the velocity controller, where $\mathbf{x}$ is the sensed position. Here, we have simply substituted the quantity $\mathbf{x}_c - \mathbf{x}$ for $\mathbf{v}_c$ in Equation 3.23.

Now, suppose instead that we have a position controller at our disposal. Given a commanded spring position $\mathbf{p}$, an interpolated trajectory is constructed, consisting of a series of closely spaced via points. For each via point $\mathbf{x}_c$, the controller sends the desired actual position

$$\mathbf{x} = \mathbf{x}_c + \mathbf{f}_r/k \tag{3.25}$$

to the position controller, where k is the stiffness constant. When a transition from contact to free space is detected, the trajectory is replanned, to prevent a large motion from being commanded. For experimental purposes, this design was implemented in the robot-level robot language AML, on an IBM 7565 robot. The results of the experiment are summarized in Section 6.3.4 and Appendix C.

Chapter 4

Verification

This chapter describes an algorithm which verifies the correctness of a class of compliant motion strategies. The algorithm relies heavily on a geometric computation called *visibility analysis*, which is used in computer graphics to compute visible scenes. The reader is advised to read Appendix A before proceeding with this chapter. The appendix provides vocabulary and algorithms relating to visibility analysis.

The basic idea of the verifier is simple. Given a compliant motion strategy, and a start region in the contact space of the robot, we compute all possible contact space termination points, in the presence of bounded sensing and control uncertainty. We then verify that the termination points are contained in the goal region. Computing the possible termination points of a commanded motion from a start region is referred to as *simulating* the motion.

4.1 Input to the Verifier

The input to the verifier consists of:

- a polyhedral environment, representing obstacles in the configuration space of a Cartesian robot which can translate in three dimensions. We will assume that the environment is bounded, and that the faces of the polyhedra are convex.
- a start region, given by a list of convex surface polygons in the environment.
- a goal region, also given by list of convex surface polygons in the environment.
- a compliant motion strategy.

Figure 4.1 shows a simple example of an environment, consisting of two blocks. We will be using this environment to illustrate the concepts of this chapter. All of the results that we will show with respect to this environment were computed by a verification program.

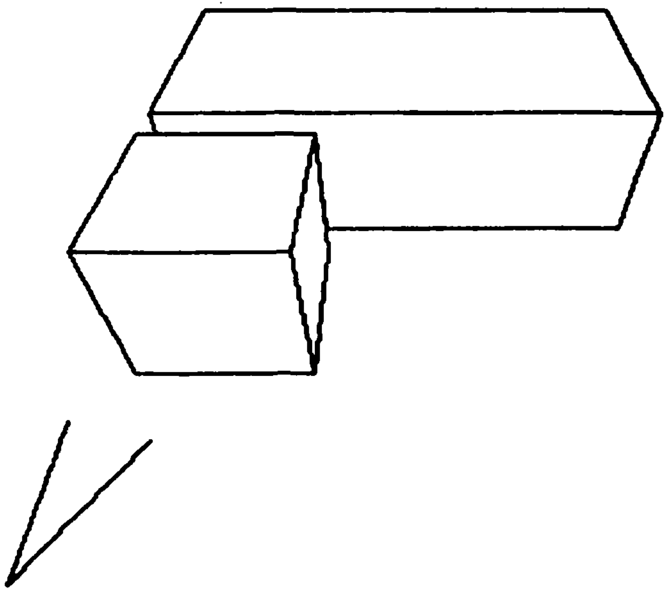


Figure 4.1: An environment consisting of two blocks. The velocity cone is shown. It's angle is .25 radians.

We introduced compliant motions in Section 1.2.2. Recall that we are interested in compliant motions which start and stop in a contact configuration. A compliant motion is specified by a commanded velocity or position, together with a set of position and force termination conditions. A commanded velocity is specified when the generalized damper trajectory model is desired. A commanded position is specified when the generalized spring trajectory model is desired. Chapter 3 discussed the properties of compliant motions under each of these trajectory models.

Compliant motion strategies were introduced in Section 1.2.3. A compliant motion strategy consists of a set of compliant motions, arranged in some execution order. A motion strategy may contain conditional tests, which offer the robot a choice of motions based on sensory feedback after a motion has been completed. Figure 4.2 shows an example of a compliant motion strategy.

4.2 Overview of the Verifier

Initially, assume that we know how to simulate a single motion. To simulate a compliant motion strategy without conditionals, we merely chain one-step simulations together. The termination points of one motion are used as the start region of the next motion. The termination points of the entire sequence are equal to the termination points of the final motion in the sequence. To verify the reliability of the sequence, we verify that all of the final termination points are contained in the goal region.

Verifying a motion strategy with conditional tests, such as that of Figure 4.2, is

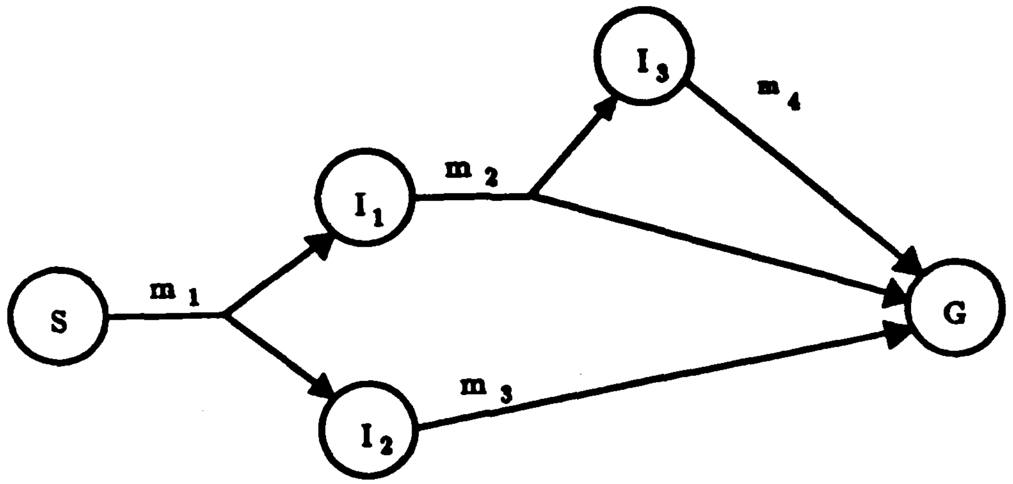


Figure 4.2: A directed graph representing a compliant motion strategy. A node in the graph represents a set of contact positions of the robot. An arc represents a commanded motion. The robot starts in set S. After issuing motion m_1, it tests whether its position and force sensors are consistent with the sets I_1 or I_2. If the sensors are consistent with I_1, then the robot issues motion m_2 next, else it issues m_3. If the strategy is reliable, then the robot eventually stops in the goal set G, despite bounded sensing and control errors.

slightly more complicated. Beginning with the first motion, we must verify geometrically that the termination points of each motion are contained in the nodes that follow the motion. For example, in Figure 4.2, the termination points of m_1 must be contained in $I_1 \bigcup I_2$. If not, then the motion strategy can fail. If they are, then all of the nodes that follow in the graph are recursively simulated, using the actual termination points as start regions. For example, in Figure 4.2, let R_1 denote the termination region of m_1. If $R_1 \subseteq I_1 \bigcup I_2$, then motion m_2 is simulated from the start region $R_1 \bigcap I_1$, and motion m_3 is simulated from the start region $R_1 \bigcap I_2$. The recursive simulation ends when a leaf node (a node with no motion arcs) is reached. If no simulated motion fails a containment test, and the final termination points are contained in the goal region, then the motion strategy is reliable.

We have thus reduced the problem of verifying a compliant motion strategy to the problems of simulating a single commanded motion, and testing for containment. Initially, we will assume that motion termination is by sticking only. We will then extend the method to sensing termination in Section 4.5.

Suppose that we are to compute the termination points of a commanded motion m from a start region S under sticking termination. We will describe a framework for performing this simulation. Then, we will discuss why the framework is correct, and describe the implementation of the framework. Here is the framework:

Algorithm 4.1 *Simulation Framework*

1. *Compute the set $F_m(S)$ of contact points that the robot can reach via straight-line trajectories from S under m.*

2. *Add to $F_m(S)$ the contact points that the robot can reach by sliding from $F_m(S)$ under m. (Note that if the robot is guaranteed to stick at a point in $F_m(S)$, then it cannot slide away from there.)*
3. *Recursively call the framework with newly reachable edges as start regions. Add to $F_m(S)$ any new reachable contact points that are returned.*
4. *If this is the top-level call, return $T_m(S)$, the subset of $F_m(S)$ at which the robot might stick.*
5. *Otherwise, return $F_m(S)$.*

This framework involves two basic tasks, (1) computing the set $F_m(S)$ of contact points that are reachable from S under m, and (2) reducing $F_m(S)$ to the set $T_m(S)$ of points at which the robot might terminate by sticking. $F_m(S)$ is called the *forward projection* of S under m. A forward projection defines a set of points which are candidates for termination. The application of a termination condition to a forward projection determines the set $T_m(S)$ of actual termination points. $T_m(S)$ is called the *simulation* of m from S, with respect to a particular termination condition. In this case, the termination condition to be applied is sticking.

The framework is described in terms of points. In implementation, one must actually deal with sets of points. Our implementation represents forward projections and simulations as lists of polygons.

4.3 Generalized Damper Simulation

Section 4.3.1 establishes the correctness of the framework under the generalized damper trajectory model. The rest of the section describes the implementation of the framework. Section 4.3.2 discusses the computation of straight-line reachable contact points when the start region is a point. Section 4.3.3 discusses the computation of straight-line reachable contact points when the start region is an edge. Section 4.3.4 discusses the computation of straight-line reachable contact points when the start region is a polygon. Section 4.3.5 discusses the computation of contact points that are reachable by sliding.

4.3.1 Correctness of the Simulation Framework

We can solidify our confidence in the framework by proving that it works under the generalized damper trajectory model. We will prove that in steps 1-3, the framework computes the forward projection correctly. Since sticking termination points are a subset of the forward projection, it should be clear that step 4 computes the simulation correctly as long as steps 1-3 work. We will assume for now that the individual steps of the framework can be implemented correctly.

Recall from Section 3.2 that the free space velocities of a robot under the generalized damper trajectory model are contained in a cone, called the velocity cone, of cone angle $\theta_c \leq \frac{\pi}{2}$ (Figure 3.1). It is assumed that a trajectory may instantaneously switch between velocities.

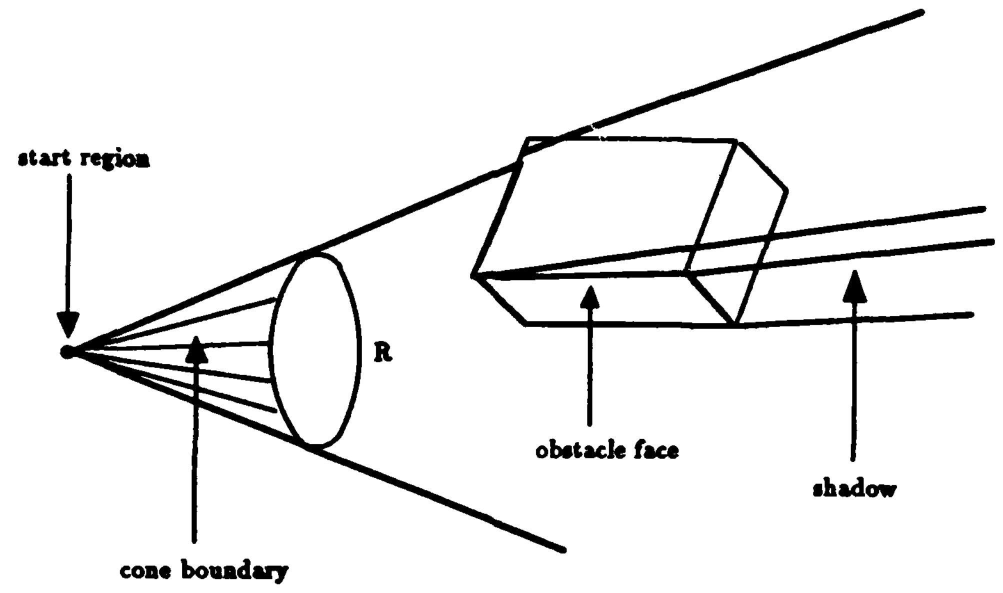


Figure 4.3: The volume R that can be traversed by straight-line trajectories from a point. R is bounded by the start point, obstacle faces, free space boundaries of the velocity cone, and free space shadows caused by straight-line reachable edges.

There are two types of reachable contact points. The first type is a contact point that is reachable by a straight-line trajectory from the start region (even though an actual trajectory may switch between velocities along the way). Figure 4.3 shows the volume R that can be reached by straight-line trajectories from a start point. In general, R is bounded by (1) start points, (2) obstacle faces, (3) free space boundaries of the velocity cone, and (4) free space shadows caused by straight-line reachable edges. A *shadow* is a surface in free space that is formed by extending rays from a start point past a straight-line reachable edge.

The second type of reachable contact point is not reachable by a straight-line trajectory, for one of two reasons:

1. A contact point may be reachable by a free space trajectory which bends around an obstacle and out of the straight-line reachable volume. (See Figure 4.4.)

2. A contact point may be reachable by a trajectory which strikes a surface, and slides out of the straight-line reachable volume. (See Figure 4.5.)

We will argue that all reachable points are equivalently reachable by a straight-line trajectory from either (a) the start region, or (b) a reachable obstacle edge. A reachable point under case (b) is computed by the recursive call in the framework. We will demonstrate the correctness of the framework by induction on the number of obstacle faces. We will begin with the base case, in which there is just one

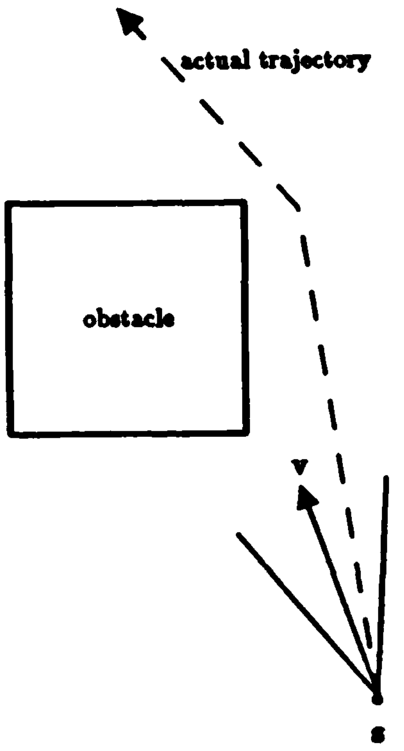


Figure 4.4: A robot trajectory may exit the straight-line reachable volume by bending around an obstacle.

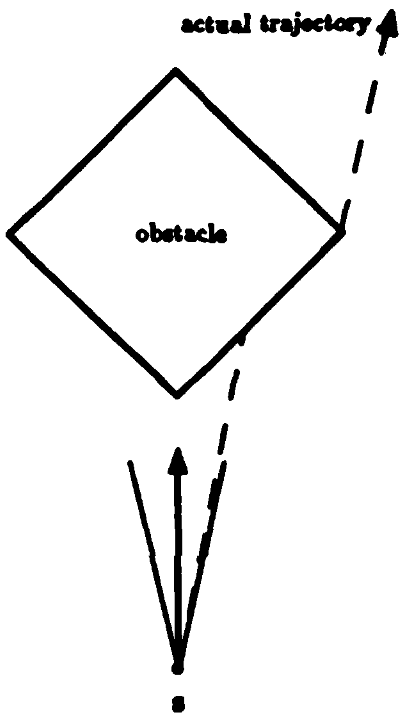


Figure 4.5: A robot trajectory may exit the straight-line reachable volume by sliding on an obstacle.

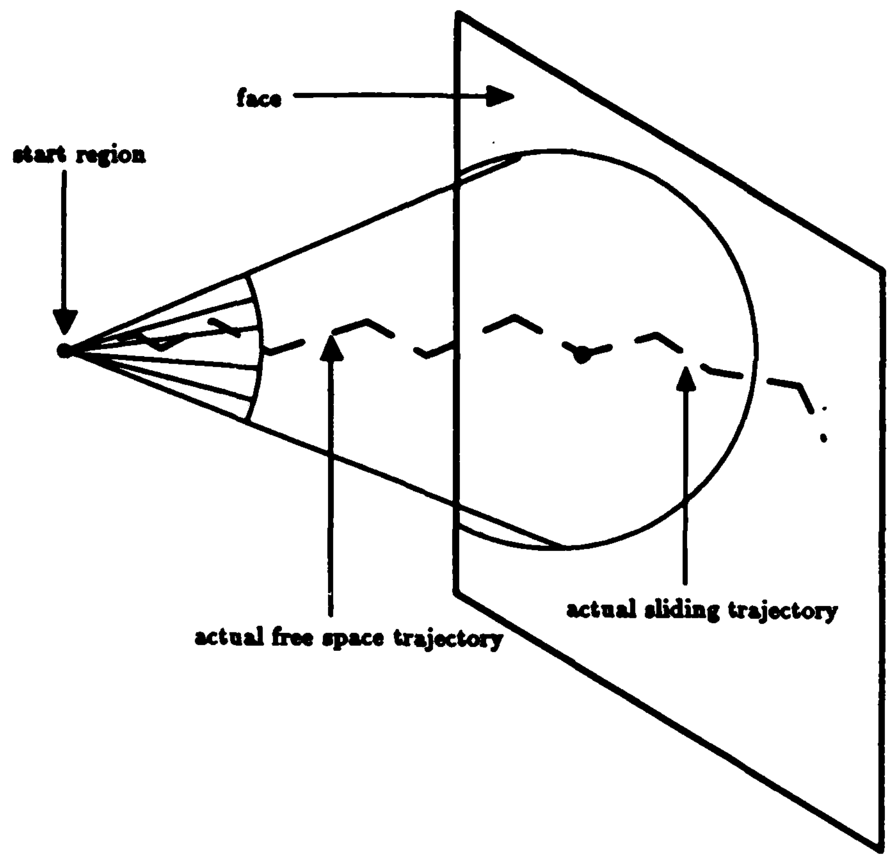


Figure 4.6: The case where there is only one obstacle face. All reachable points are equivalently reachable by a combination of a straight-line free space trajectory and a sliding trajectory.

obstacle face. Since velocity cones are convex, all reachable contact points are equivalently reachable by a combination of a straight-line free space trajectory and a sliding trajectory. This is true despite the fact that an actual trajectory may consist of a number of different velocities, as shown in Figure 4.6. Steps 1 and 2 of the framework thus compute all reachable contact points for the base case.

As the induction hypothesis, we now assume that the framework is correct when there are n obstacle faces, for $n \geq 1$. Suppose that there are $n + 1$ obstacle faces. Consider the volume R that can be traversed by straight-line trajectories from the start region (Figure 4.3). The reachable contact points within R are precisely its bounding obstacle faces. These faces are computed by step 1 of the framework. We now have to show that the framework computes all reachable contact points that are outside of R as well. For the robot to travel to a surface outside of R, it has to pass through the boundary of R. Thus, we need only consider trajectories that start at the boundary of R. We consider each type of boundary point in turn:

- *Start points.* The only way that the robot can exit R from a start point is by sliding. Step 2 of the framework computes these reachable points.

- *Free space boundaries of the velocity cone.* The robot cannot exit R from this boundary. To do so would require a velocity not contained in the velocity cone.
- *Free space shadows.* A shadow is reachable from the edge that causes it. Thus, all contact points that are reachable from a shadow are also reachable from the edge that causes the shadow. After the robot leaves an edge, the face that the edge is on is no longer reachable, since $\theta_v \leq \frac{\pi}{2}$. Thus, there are at most n reachable faces from the edge. By the induction hypothesis, step 3 of the framework correctly computes all contact points that are reachable from the edge. In the process, all contact points that are reachable from the shadow are computed. Note that this computation returns contact points that are reachable by bending around an obstacle.
- *Obstacle faces.* There are two ways that the robot can exit R from an obstacle face:

    1. By sliding along the face to a point on the face but not in R. Step 2 of the framework computes these reachable points.
    2. By exiting R from a reachable edge of the face. After the robot leaves the edge, there are at most n faces left to reach. By the induction hypothesis, step 3 of the framework correctly computes all contact points that are reachable from the edge.

We have shown that the framework computes all contact points that are reachable within R and from the boundary of R. Thus, it computes all reachable contact points. ∎

4.3.2 Reachable Points From a Point

The first step of the simulation framework is to compute the contact points that are reachable by straight-line trajectories from the start region. There are several possible types of start regions. Consider first the case where the start region is a point. The commanded velocity is $\mathbf{v}$, with velocity uncertainty θ_v.

A natural tool for this computation is convex visibility analysis (Section A.2). Convex visibility analysis computes all contact points that can be seen by a viewer looking along a viewing axis with a given viewing angle. Think of the start point of the robot as the viewer, $\mathbf{v}$ as the viewing axis, θ_v as the viewing angle, and the velocity cone as the viewing volume.

Figure 4.7 shows an example of straight-line reachability from a point, using the double block environment of Figure 4.1.

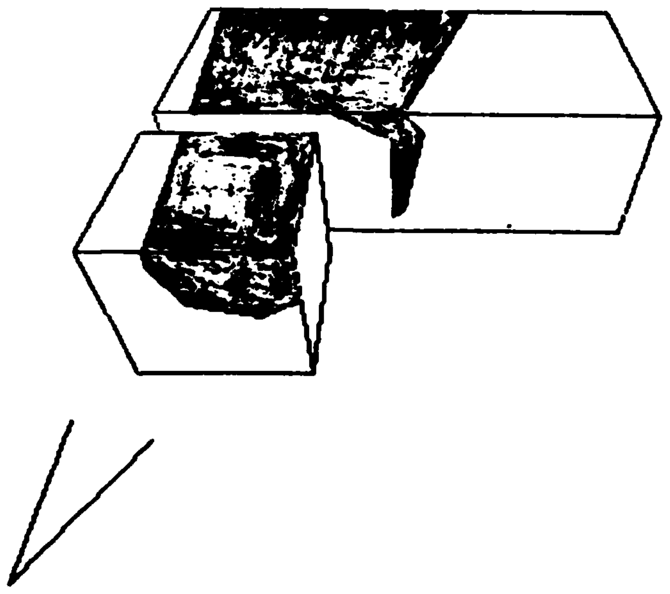


Figure 4.7: An example of straight-line reachability from a point. The start point is located at the vertex of the indicated velocity cone. The shaded regions shown here contain the contact points that are reachable by straight-line trajectories from the start point.

4.3.3 Reachable Points From an Edge

Figure 4.8 shows the reachable volume when the start region is an edge $\bar{e}$. The volume is the union of a wedge and two cones. The cones contain points that are reachable from the endpoints of $\bar{e}$. The wedge adds additional points that are reachable from the interior of $\bar{e}$.

A naive algorithm to compute straight-line reachability within this volume would be to discretize $\bar{e}$, and compute the union of the reachable points from each discrete edge point, using convex visibility analysis. A much better algorithm returns the union of only three visibility computations, two from the endpoints of $\bar{e}$, and one from the interior. The endpoint computations are performed by convex visibility analysis. The interior computation is performed by *y-convex visibility analysis*, which is similar to convex visibility analysis, but with two differences. First, the reachable volume is the wedge shown in Figure 4.8, rather than a cone. Second, a different transformation is applied to the vertices of the environment, rather than the convex perspective transformation. This transformation will be described below.

The algorithm for computing straight-line reachability from an edge is as follows:

Algorithm 4.2 *Straight-line Reachability From an Edge (Generalized Damper)*

1. *Compute the set F of contact points that are straight-line reachable from the endpoints of $\bar{e}$, using convex visibility analysis.*

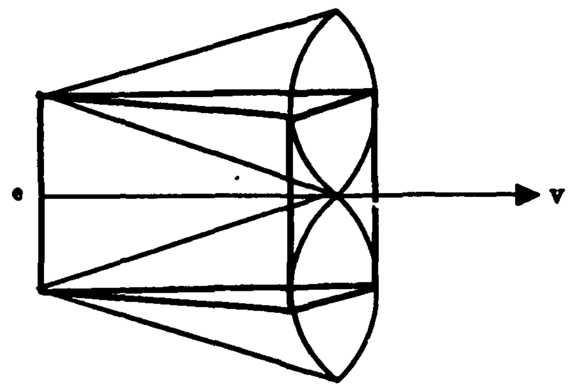


Figure 4.8: The commanded velocity is **v**. The start region is the edge $\bar{e}$. The reachable volume is equal to the union of a wedge and two cones.

2. *Add to F the contact points that are returned by y-convex visibility analysis from $\bar{e}$.*
3. *Recursively call the algorithm, using newly reachable edges as start edges. Add to F any new reachable contact points that are returned.*
4. *Return F.*

We said above that the algorithm would require only three visibility computations. Algorithm 4.2 would seem to require more than three, since it calls itself recursively. However, the algorithm is called by the simulation framework (Algorithm 4.1), which also calls itself recursively using newly reachable edges as start edges. Step 3 can actually be left out of Algorithm 4.2, under the assumption that the calling algorithm will perform this chore.

Y-convex Visibility Analysis

Consider Figure 4.9, which shows the coordinate system formed by $\bar{e}$ and **v**. The origin of the coordinate system is placed at the rearmost endpoint of $\bar{e}$ along **v**. **q** is the other endpoint of $\bar{e}$. The viewing axis (the negative z-axis) is directed along **v**. The x-axis is placed perpendicular to the negative z-axis. (Its orientation about the z-axis will be discussed below.) Adding an appropriate y-axis completes the right-handed viewer coordinate system for the computation.

The purpose of y-convex visibility analysis is to return all contact points that are straight-line reachable from $\bar{e}$ by trajectories which have uncertainty in the y direction but not in the x direction. Shortly, we will argue that Algorithm 4.2 works correctly, despite the fact that it seems to ignore x uncertainty. It turns out that the algorithm simulates x uncertainty implicitly by issuing the recursive calls.

In order to perform y-convex visibility analysis, we need a start edge which is perpendicular to **v**. This is because the image is obtained from the whole edge, not just a point, and the image plane must be perpendicular to the viewing axis. We will now show how to modify $\bar{e}$ so that it is perpendicular to **v**.

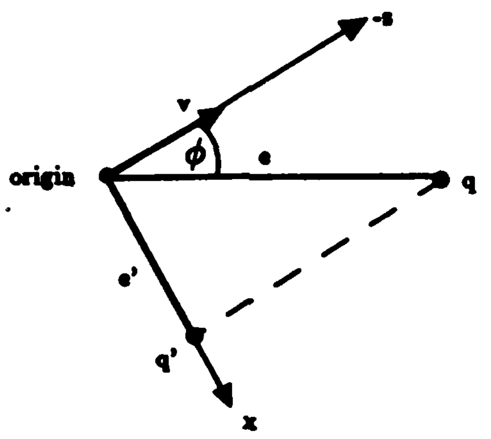


Figure 4.9: The coordinate system formed by the start edge $\bar{e}$ and the commanded velocity **v**. The origin is placed at the rearmost endpoint of $\bar{e}$ along **v**. **q** is the other endpoint of $\bar{e}$. The negative z-axis is directed along **v**. The x-axis is placed perpendicular to the negative z-axis. (Its orientation about the z-axis is not shown in the figure.) $\overline{e'}$ is the projection of $\bar{e}$ onto the x-axis.

Instead of $\bar{e}$, the start edge will be the projection $\overline{e'}$ of $\bar{e}$ onto the x-axis. This works only if the obstacle of which $\bar{e}$ is a part does not occlude the view from $\overline{e'}$ any more than it does from $\bar{e}$. This condition can be guaranteed by a careful choice of the orientation of the x-axis about the z-axis. The orientation is chosen by picking the point **q**$'$ in Figure 4.9 such that the point **q** gets placed on the boundary of the velocity cone emanating from **q**$'$. The cone from **q**$'$ thus contains the cone from **q**. There are two possible choices for **q**$'$; it is picked such that the obstacle containing $\bar{e}$ is exterior to the cone from **q**$'$ if possible.

Although using $\overline{e'}$ rather than $\bar{e}$ does not affect the correctness of the computation, it does affect the completeness. Some additional contact points which are not actually reachable from $\bar{e}$ may be returned. This has a conservative effect on the verification. The efficiency of the computation makes this conservatism worthwhile.

We are now ready to give the details of the transformation used in y-convex visibility analysis. Recall that we want to simulate uncertainty along the y-axis but not along the x-axis. Y-convex visibility analysis does this by applying the convex perspective transformation to the y-coordinates of the vertices, while leaving the x-coordinates alone (i.e. orthographically transforming them). This transformation is called the *y-convex perspective transformation*, and produces faces in the image plane whose boundary edges are parabolic. The reason for this is that the original y-coordinate of a vertex is replaced by the term $fy/(-z)$ (Equation A.2). This term is quadratic because it has two independent variables, y and z. The x-coordinate is not modified to preserve linearity, as it was in the convex perspective transformation. To simplify the clipping operations that are used in visibility analysis, the projected faces can be approximated by polygons.

Figure 4.10 shows an example of straight-line reachability from an edge, using the double block environment of Figure 4.1.

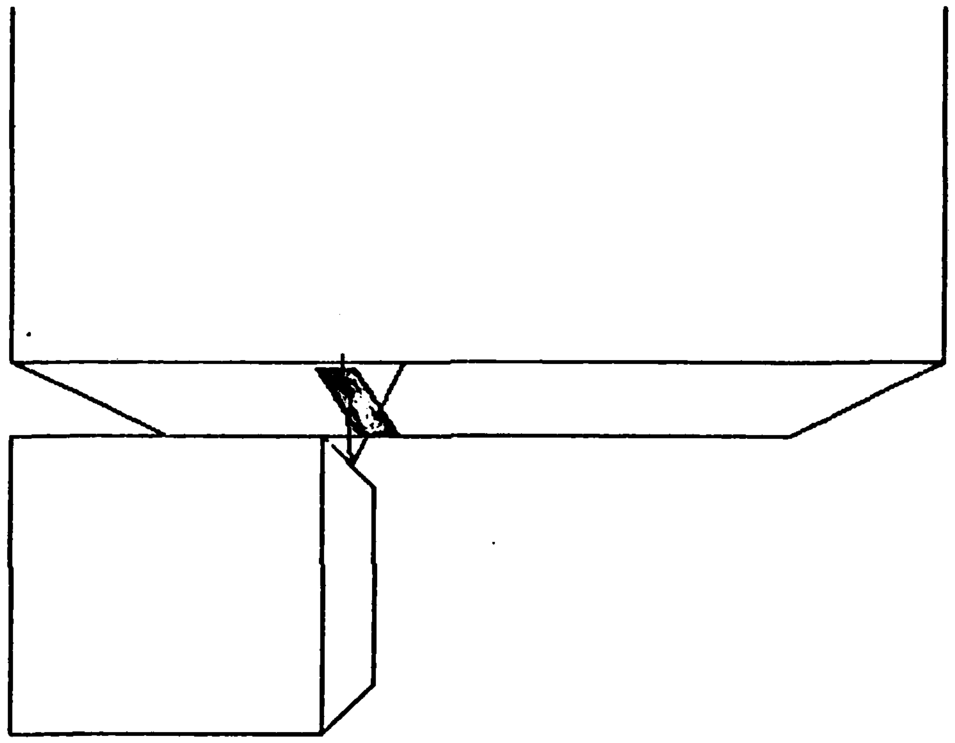


Figure 4.10: An example of straight-line reachability from an edge. The start edge is located at the vertex of the velocity cone, as shown. The velocity uncertainty angle θ_v is equal to .25 radians. The shaded region shown here contains the contact points that are reachable by straight-line trajectories from the start edge.

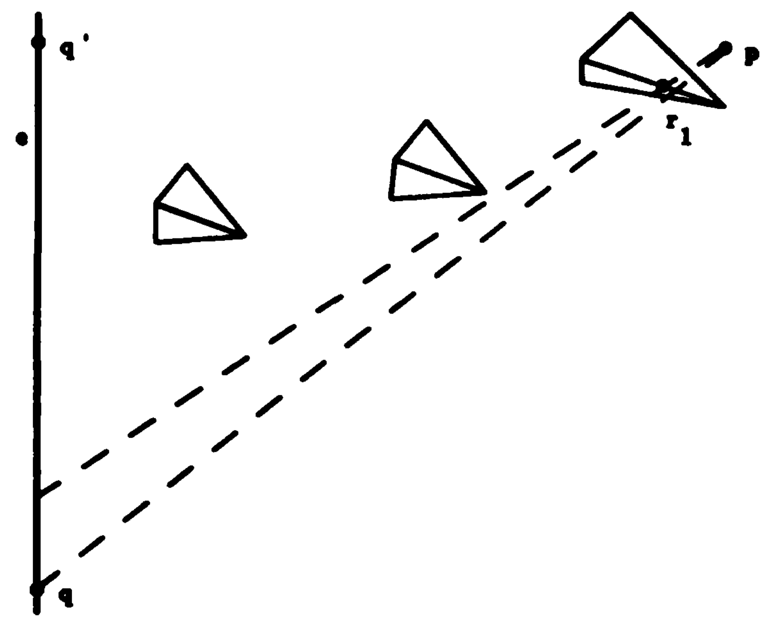


Figure 4.11: We slide a tight string along the edge from **q** towards **q**′ until contact is made with the environment.

Correctness of the Edge Algorithm

The purpose of Algorithm 4.2 is to compute all contact points that are reachable by straight-line trajectories from an edge $\bar{e}$. If $\bar{e}$ is not perpendicular to **v**, then we can modify it appropriately, as described above. We will thus assume that $\bar{e}$ is perpendicular to **v**.

The algorithm consists of (1) applying convex visibility analysis from the endpoints of $\bar{e}$, (2) applying y-convex visibility analysis from $\bar{e}$, and (3) recursively calling the algorithm from all newly reachable edges.

All contact points that are straight-line reachable from $\bar{e}$ are contained in the union of a wedge and two cones, as shown in Figure 4.8. Let **p** be a point which is straight-line reachable from a point **q** on $\bar{e}$. If **p** is in one of the endpoint cones, then convex visibility analysis from an endpoint will return **p**. If $\overline{\mathbf{pq}}$ is perpendicular to $\bar{e}$, then $\overline{\mathbf{pq}}$ corresponds to a trajectory which has no error in the x direction, and y-convex visibility analysis from $\bar{e}$ will return **p**. The remaining case is when **p** is in the wedge but not in an endpoint cone, and $\overline{\mathbf{pq}}$ is not perpendicular to $\bar{e}$.

Let **q**′ be the point on $\bar{e}$ where $\overline{\mathbf{pq}'}$ is perpendicular to $\bar{e}$, as shown in Figure 4.11. **q**′ exists, since **p** is in the wedge. Suppose that we connect **p** and **q** by a tight string. Then, we slide the string along $\bar{e}$ toward **q**′, keeping it tight, until the string contacts an obstacle between $\bar{e}$ and **p**. The string cannot make it all the way to **q**′, for if it did then **p** would have been returned by y-convex visibility analysis. Thus, a contact with the environment is guaranteed.

Let $\mathbf{r}_1$ be the point in the environment where the string makes contact. (See Figure 4.11.) Since $\mathbf{r}_1$ is straight-line reachable from $\bar{e}$, we have created a sub-

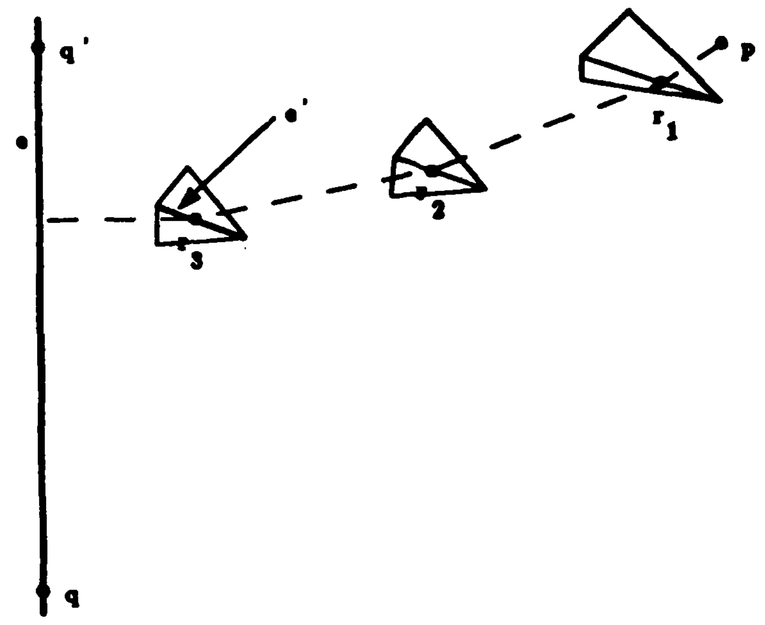


Figure 4.12: We continue to find closer contact points to $\overline{e}$, until the string becomes perpendicular to $\overline{e}$.

problem of the same form as the original problem, with $\mathbf{r}_1$ substituted for $\mathbf{p}$. The distance from $\overline{e}$ to $\mathbf{r}_1$ along the string is smaller than the distance from $\overline{e}$ to $\mathbf{p}$. We can find successively closer contact points $\mathbf{r}_2$, $\mathbf{r}_3$, ..., $\mathbf{r}_k$ to $\overline{e}$ in this fashion, until eventually the string from $\overline{e}$ to $\mathbf{r}_k$ will become perpendicular to $\overline{e}$, as illustrated in Figure 4.12. This is guaranteed to happen, since the distance from $\overline{e}$ to the contact point is successively decreasing, and we will eventually run out of obstacles between $\overline{e}$ and the contact point.

We now have a connected chain of strings between $\overline{e}$ and $\mathbf{p}$, for which the first link is perpendicular to $\overline{e}$. Because of this perpendicularity, y-convex visibility analysis from $\overline{e}$ can return an edge segment $\overline{e'}$ containing $\mathbf{r}_k$. (See Figure 4.12.) The second link in the chain, from $\overline{e'}$ to $\mathbf{r}_{k-1}$, is a subproblem of the same form as the original problem involving $\overline{e}$ and $\mathbf{p}$. Just as we did with the original problem, we can replace this link by a subchain, in which the first sublink is either perpendicular to $\overline{e'}$, or attached to $\overline{e'}$ at an endpoint. In either case, y-convex visibility analysis can return the first contact point in the subchain. We now have a chain between $\overline{e}$ and p in which the first two contact points are computable by y-convex visibility analysis. This process can be continued until all of the contact points in the chain from $\overline{e}$ to p are computable by y-convex visibility analysis. Since Algorithm 4.2 calls itself recursively using newly reachable edges as start edges, it will work its way down the chain from $\overline{e}$ until eventually $\mathbf{p}$ is returned.

Since the algorithm returns $\mathbf{p}$, we can conclude that it returns all straight-line reachable points. ∎

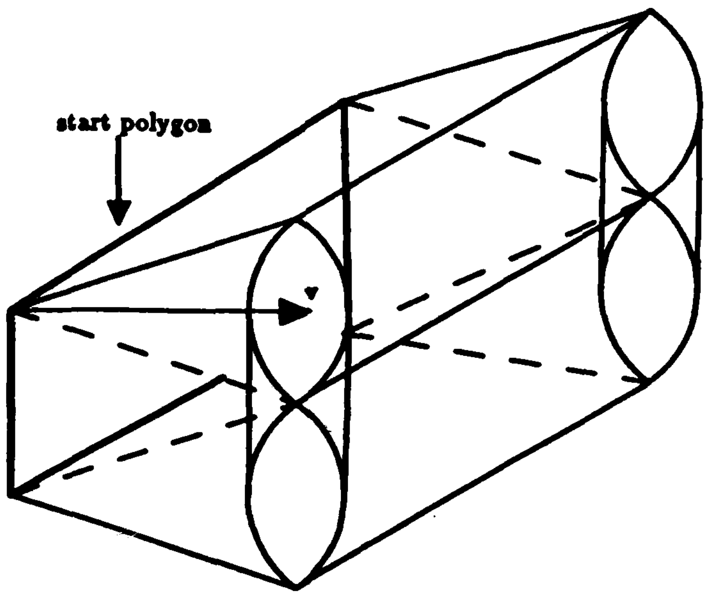


Figure 4.13: The commanded velocity is v. The start region is a polygon. The reachable volume from the edges of the start polygon is shown.

4.3.4 Reachable Points From a Polygon

Now consider the case where the start region is a polygon. We begin by computing straight-line reachable points from the bounding edges of the polygon, using Algorithm 4.2, as shown in Figure 4.13. This leaves a volume bounded by the start polygon, some wedges, and some cones, for which reachability has not been computed. Straight-line reachable points in this bounded volume can be computed by orthographic visibility analysis (Section A.1) in the reachable volume shown in Figure 4.14. The union of all of these reachable points is returned. The proof that this algorithm works is very similar to the proof for an edge start region.

4.3.5 Sliding and Sticking

This subsection discusses the simulation of sliding, and the computation of contact points at which the robot might stick. Sliding and sticking under the generalized damper trajectory model depends on the direction of the robot's velocity in relation to the orientation of the surface, and the surface's friction cone. If the velocity is directed into the negative friction cone of the surface, then the robot will stick. If the velocity is directed outside of the negative friction cone, then the robot will slide along the projection of the velocity onto the surface. With velocity uncertainty, it is sometimes impossible to predict whether the actual velocity corresponding to a commanded velocity will be into the negative friction cone or not. Figure 3.5 shows commanded velocities which ensure sticking and sliding. Other commanded velocities may do either.

Each point of a particular face, edge, or vertex has the same friction cone. This

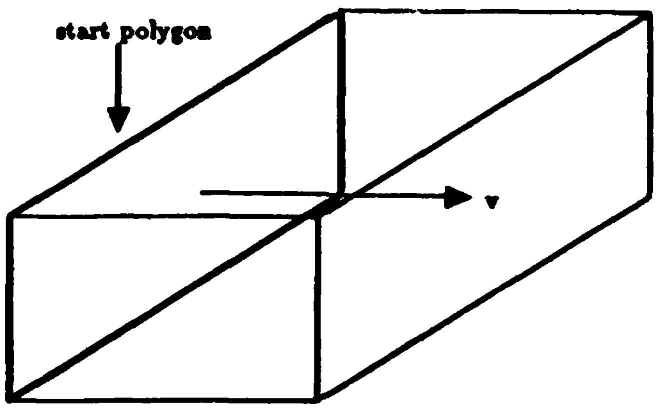


Figure 4.14: The commanded velocity is **v**. The start region is a polygon. The reachable volume from the start polygon under orthographic visibility analysis is shown.

means that if sliding is possible at a point of a particular face, edge, or vertex, then it is possible anywhere on it. For a given commanded velocity, we can compute whether sliding is possible on a face, edge, or vertex by subtracting its negative friction cone from the velocity cone. If the difference is nonempty, then sliding is possible. Similarly, we can compute whether sticking is possible on a face, edge, or vertex by intersecting its negative friction cone with the velocity cone. If the intersection is nonempty, then sticking is possible.

If sliding is possible, then the sliding trajectory on a face, edge or vertex is given by projecting the sliding velocities onto the surface. Since edges and vertices are boundaries of faces, it suffices to compute sliding trajectories on faces only. Note that we have to compute sticking on edges and vertices individually. Edges and vertices usually have larger friction cones that their cobounding faces, and thus can sometimes cause sticking when their cobounding faces don't.

We will now show how to compute a sliding trajectory on a face F. First, we need some new terminology:

- ϕ is the friction cone angle of F.
- $\mu = \tan\phi$ is the coefficient of friction of F.
- f is the plane containing F.
- $\mathbf{p}$ is a point on F.
- $\mathbf{n}$ is the outward unit normal vector to F.
- $cone(\mathbf{v}, \theta)$ is a cone whose vertex is at the origin, whose central axis is given by $\mathbf{v}$, and whose cone angle is θ.
- $ext(\mathbf{n})$ is the halfspace of $\Re^3$ which is exterior to the plane through the origin which has an outward unit normal vector $\mathbf{n}$.

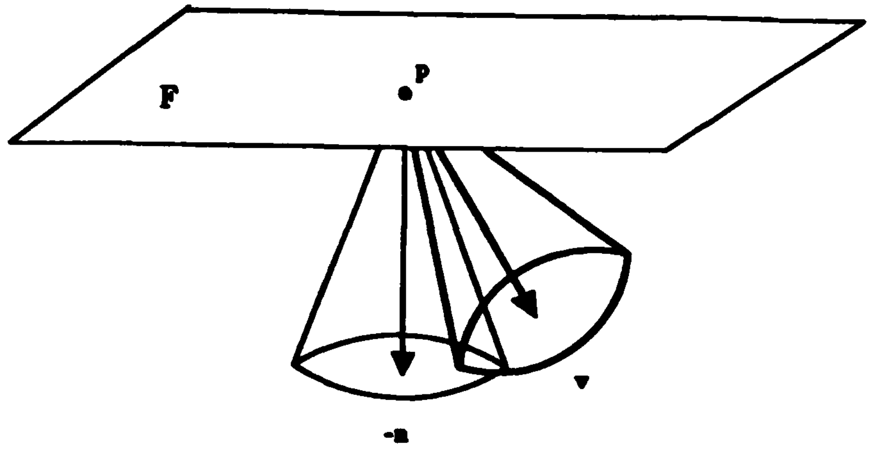


Figure 4.15: If the robot were located at point p on face F, then its velocity cone and the negative friction cone of the face might look like this.

Consider Figure 4.15. The set of sliding velocities Γ is equal to

$$\Gamma = cone(\mathbf{v}, \theta_v) - cone(-\mathbf{n}, \phi) - ext(\mathbf{n}). \tag{4.1}$$

If Γ is empty, then there is no possibility of sliding on F. To compute whether Γ is empty, we can project the geometry of Figure 4.15 onto the plane which contains $\mathbf{v}$ and $-\mathbf{n}$, as shown in Figure 4.16. This projection is performed such that $\mathbf{v}$ projects to the right of $-\mathbf{n}$. $\mathbf{f}_c$ and $\mathbf{v}_c$ are extensions of $-\mathbf{n}$ and $\mathbf{v}$, respectively. $\mathbf{f}_l$ and $\mathbf{f}_r$ are the left and right extreme vectors of the projected friction cone, respectively. $\mathbf{v}_l$ and $\mathbf{v}_r$ are the left and right extreme vectors of the projected velocity cone, respectively. We can intersect the six vectors $\mathbf{f}_l$, $\mathbf{f}_c$, $\mathbf{f}_r$, $\mathbf{v}_l$, $\mathbf{v}_c$, and $\mathbf{v}_r$ with a horizontal line l one unit below the plane f, as shown in Figure 4.16. The length of all of these vectors is determined by the point at which they intersect l. Let f_l, f_c, f_r, v_l, v_c, and v_r be the horizontal components of the intersection points along l, respectively. Then Γ is empty if and only if either of the following conditions hold:

- The friction cone angle is equal to $\frac{\pi}{2}$.
- The velocity cone is completely contained in the friction cone ($v_l \geq f_l$ and $v_r \leq f_r$).

There is a possibility of sticking on S if and only if the quantity

$$cone(\mathbf{v}, \theta_v) \bigcap cone(-\mathbf{n}, \phi) \tag{4.2}$$

is nonempty. This quantity is nonempty if and only if $v_l \leq f_r$.

Once we have determined that the robot may slide on F, we can compute the sliding trajectories from S. The *sliding cone* is the planar cone of sliding velocities which result from projecting Γ onto the plane f. (See Figure 4.17.) The sliding trajectories from S are equal to the union of the sliding cones that originate at all

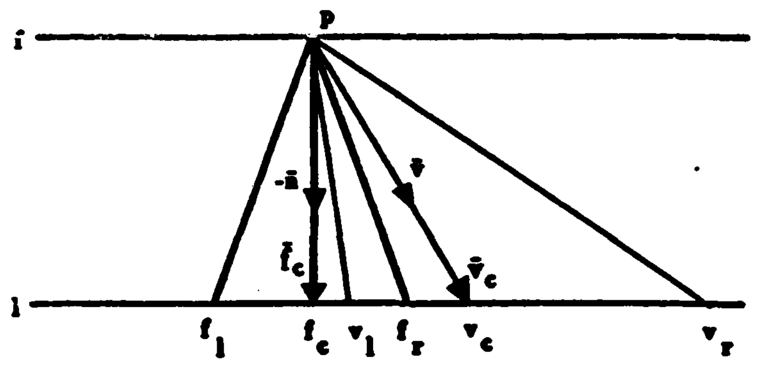


Figure 4.16: The sliding geometry, projected onto the plane containing **v** and $-$**n**. $\mathbf{f}_c$ and $\mathbf{v}_c$ are extensions of $-$**n** and **v**, respectively. $\mathbf{f}_l$ and $\mathbf{f}_r$ are the left and right extreme vectors of the projected friction cone, respectively. $\mathbf{v}_l$ and $\mathbf{v}_r$ are the left and right extreme vectors of the projected velocity cone, respectively. l is a horizontal line one unit below f.

of the points in S, intersected with F, as shown in Figure 4.18. The sliding region can be computed by the following algorithm:

Algorithm 4.3 *Sliding Simulation (Generalized Damper)*

1. *If the friction cone is contained inside the velocity cone ($v_l < f_l$ and $v_r > f_r$), then the robot can slide in any direction, and all of F should be returned.*
2. *Compute the sliding cone angle and the bounding vectors* $\mathbf{b}_l$ *and* $\mathbf{b}_r$ *of the sliding cone. (See below.) If the sliding cone angle is greater than $\frac{\pi}{2}$, then the robot can slide in any direction, and all of F should be returned. (See Figure 4.19.)*
3. *Connect the left bounding vector* $\mathbf{b}_l$ *of the sliding cone to the left support vertex of the polygon S with respect to the line containing* $\mathbf{b}_l$. *The left support vertex of a polygon P with respect to a line l is the first vertex of P to contact l when l is moved toward P from the left.*
4. *Connect the right bounding vector* $\mathbf{b}_r$ *of the sliding cone to the right support vertex of the polygon S with respect to the line containing* $\mathbf{b}_r$.
5. *Extend* $\mathbf{b}_l$ *and* $\mathbf{b}_r$ *until they intersect the boundary of the face F.*
6. *Return the polygonal sliding region R bounded by* $\mathbf{b}_l$, *some boundary edges of F,* $\mathbf{b}_r$, *and some boundary edges of S. (See Figure 4.18.)*

Computing the Sliding Cone

The sliding cone can be parameterized by its cone axis and angle. It is an easy matter to compute the extreme cone vectors $\mathbf{b}_l$ and $\mathbf{b}_r$, given the cone axis and angle.

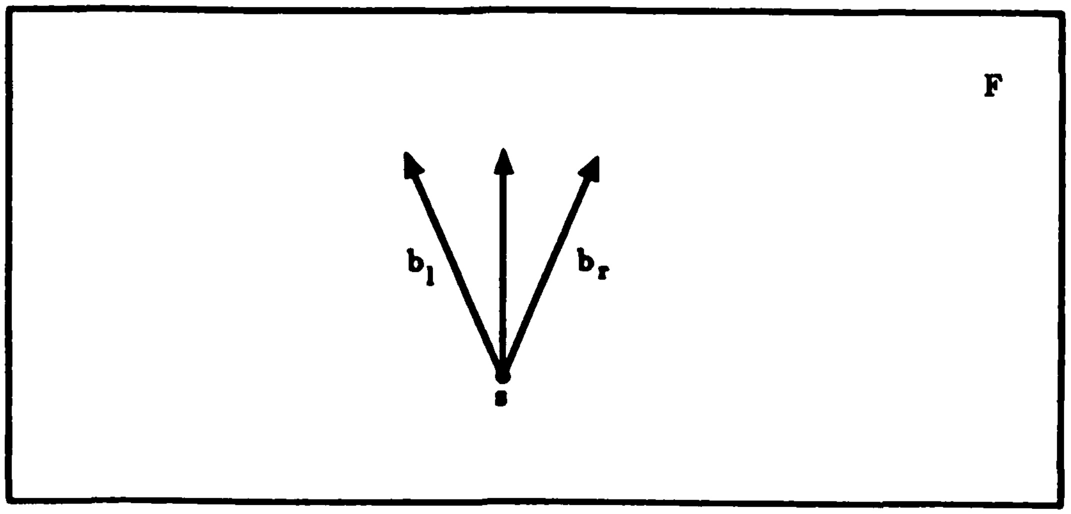


Figure 4.17: This is a top view of the sliding face F, showing the cone of sliding velocities from a point s. The sliding velocities are bounded by the vectors $\mathbf{b}_l$ and $\mathbf{b}_r$.

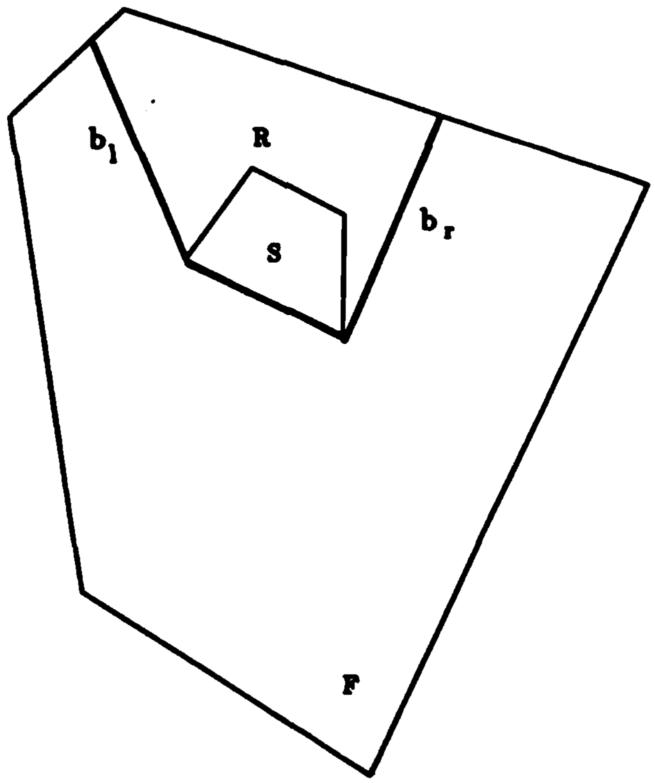


Figure 4.18: This is a top view of the sliding face F, showing the sliding region R from a start polygon S. R is bounded by $\mathbf{b}_l$, some boundary edges of F, $\mathbf{b}_r$, and some boundary edges of S.

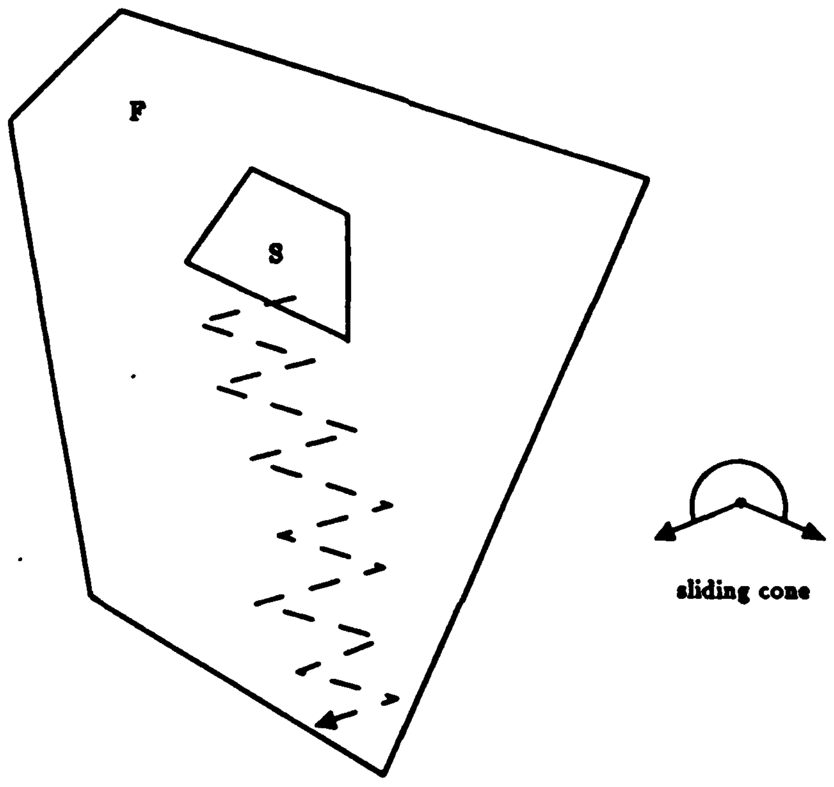


Figure 4.19: An example in which the sliding cone angle is greater than $\frac{\pi}{2}$. A possible sliding trajectory of the robot is shown.

The sliding cone axis is equal to the projection of $\mathbf{v}$ onto f,

$$\mathbf{v} - (\mathbf{v} \bullet \mathbf{n})\mathbf{n}. \tag{4.3}$$

Computing the sliding cone angle depends on whether the velocity cone intersects the friction cone. If the cones do not intersect nontrivially ($v_l \geq f_r$), then the sliding cone angle is determined by projecting the velocity cone onto f. The magnitude of the projection of $\mathbf{v}_c$ onto the surface is $v_c - f_c$. (See Figure 4.16.) The radius of the cone at $\mathbf{v}_c$ is $\|\mathbf{v}_c\| \tan \theta_v$. Thus, the sliding cone angle is equal to

$$\arctan(\|\mathbf{v}_c\| \tan \theta_v, v_c - f_c). \tag{4.4}$$

If the friction cone and the velocity cone intersect, then things get much more difficult. Figure 4.20 shows a cross section of the two intersecting cones, in the plane parallel to f and 1 unit below it. This figure applies only to the case where the velocity cone is completely below the plane f. The origin of the coordinate system is placed at the point where the friction cone axis intersects the cross sectional plane. The cross section of the friction cone is a circle of radius μ, centered at the origin. The cross section of the velocity cone is an ellipse. The leftmost point on the ellipse is $\mathbf{e}_l = (v_l - f_c, 0)$. The rightmost point is $\mathbf{e}_r = (v_r - f_c, 0)$. The major axis has a length of $\|\mathbf{e}_r - \mathbf{e}_l\| = v_r - v_l$. The topmost point on the ellipse is $\mathbf{e}_t = (v_c - f_c, \|\mathbf{v}_c\| \tan \theta_v)$. The foci are at $\mathbf{f}_1 = (f_1, 0)$ and $\mathbf{f}_2 = (f_2, 0)$, where f_1 and f_2 are unknowns. These unknowns can be computed by solving the following ellipse equations together:

$$\|\mathbf{e}_l - \mathbf{f}_1\| + \|\mathbf{e}_l - \mathbf{f}_2\| = 2(v_r - v_l) \tag{4.5}$$

$$\|\mathbf{e}_t - \mathbf{f}_1\| + \|\mathbf{e}_t - \mathbf{f}_2\| = 2(v_r - v_l) \tag{4.6}$$

The lengthy solutions for f_1 and f_2 are omitted here. Given the foci of the ellipse, we can compute the points where the circle and ellipse intersect. The intersection points are given by $\mathbf{i}_1 = (i_x, i_y)$ and $\mathbf{i}_2 = (i_x, -i_y)$, where i_x and i_y are unknowns. The unknowns can be computed by solving the following ellipse and circle equations simultaneously:

$$\|\mathbf{i}_1 - \mathbf{f}_1\| + \|\mathbf{i}_1 - \mathbf{f}_2\| = 2(v_r - v_l) \tag{4.7}$$

$$i_x^2 + i_y^2 = \mu^2 \tag{4.8}$$

Again, the lengthy solutions for i_x and i_y are omitted here.

If the velocity cone is partly above the plane f, then its cross section is a parabola. The vertex of the parabola is at e_l. e_t is also on the parabola. The distance a to the focus of the parabola can be computed by solving the following parabola equation:

$$(\|\mathbf{v}_c\| \tan \theta_v)^2 = 4a(v_c - v_l) \tag{4.9}$$

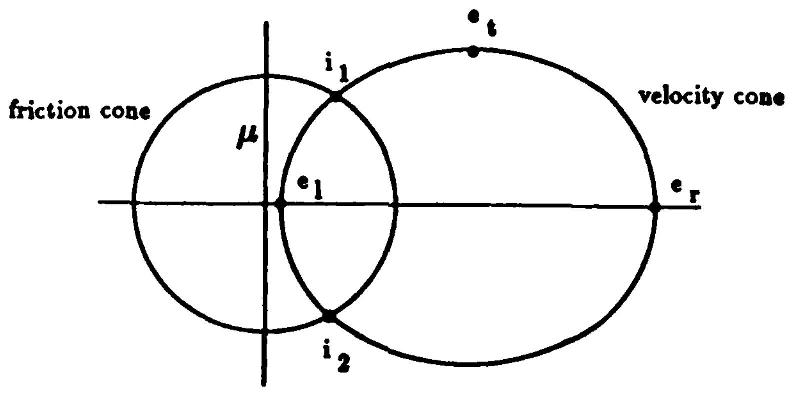


Figure 4.20: This is a cross section of the friction cone, showing its intersection with the velocity cone, for the case where the velocity cone is completely below the plane f. The origin of the coordinate system is placed at the point where the friction cone axis intersects the cross sectional plane. The cross section of the friction cone is a circle of radius μ centered at the origin. The cross section of the velocity cone is an ellipse, as shown.

The intersection point can now be computed by solving the following parabola and circle equations simultaneously:

$$i_y^2 = 4a(i_x - (v_l - f_c)) \tag{4.10}$$

$$i_x^2 + i_y^2 = \mu^2 \tag{4.11}$$

Once the intersection points have been computed, the sliding cone angle can be computed as

$$\arctan(i_y, i_x). \tag{4.12}$$

Although $\theta_v \leq \frac{\pi}{2}$, the sliding cone angle may still be greater than $\frac{\pi}{2}$. This occurs when i_y is negative. For this to occur, θ_v must be larger than ϕ. A sliding cone angle greater than $\frac{\pi}{2}$ implies that the robot can slide backwards, which means that it can slide anywhere on the face F. (See Figure 4.19.)

A Sliding Example

Consider Figure 4.7, which shows the points which are reachable by straight-line trajectories from a start point in the double block environment of Figure 4.1. The velocity uncertainty angle θ_v is equal to .25 radians. The coefficient of friction of both blocks is equal to .25. Figure 4.21 shows the contact points that can be reached by sliding from the reachable regions of Figure 4.7.

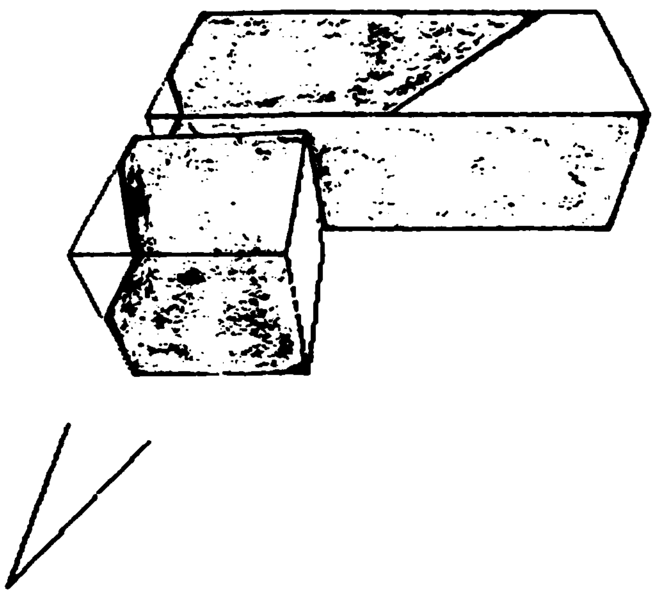


Figure 4.21: An example of a sliding simulation. The start regions are the shaded regions shown in Figure 4.7. The velocity uncertainty angle θ_v is equal to .25 radians. The coefficient of friction of both blocks is equal to .25. The shaded regions shown here contain the contact points that are reachable by sliding.

4.3.6 A Simulation Example

Figure 4.22 shows the termination points of the double block environment of Figure 4.1. Intermediate calculations in the simulation were shown in Figures 4.7, 4.10, and 4.21.

4.4 Generalized Spring Simulation

The generalized damper simulation was implemented as a service routine of the teaching system (Chapter 5). For teaching generalized spring motions, a corresponding simulation was not necessary. As a result, a corresponding simulation was never completely implemented for the generalized spring trajectory model. Nevertheless, the simulation framework (Algorithm 4.1) can be adapted quite easily to the generalized spring trajectory model. For reference, we propose an algorithm in this section. Section 4.4.1 discusses the computation of straight-line reachable contact points when the start region is a point. Section 4.4.2 discusses the computation of straight-line reachable contact points when the start region is an edge. Section 4.4.3 discusses the computation of straight-line reachable contact points when the start region is a polygon. Section 4.4.4 discusses the computation of contact points that are reachable by sliding.

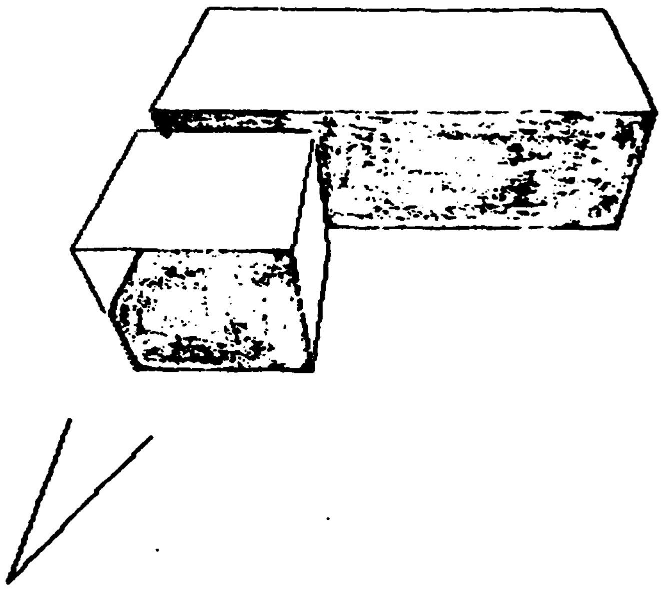


Figure 4.22: The shaded regions contain the termination points of this simulation problem.

4.4.1 Reachable Points From a Point

Suppose that the robot is in configuration s, and is assigned a commanded position **p**. Figure 3.13 shows the cylinder of free space trajectories that can result. The cylinder contains all points that are within the trajectory error ϵ_t of the commanded trajectory (Equation 3.13). Straight-line reachable points within this cylinder can be computed by convex visibility analysis. The only difference between this computation and the generalized damper analog (Section 4.3.2) is the reachable volume. The methods are otherwise identical, because both trajectory models are based on first-order dynamic systems.

4.4.2 Reachable Points From an Edge

Now suppose that the robot is located on an edge $\bar{e}$, and is assigned a commanded position **p**. Figure 4.23 shows the volume of free space trajectories that can result. This volume contains all possible commanded trajectories, and all points that are within the trajectory error ϵ_t of the commanded trajectories.

Straight-line reachable points from $\bar{e}$ can be computed by the same method used in the generalized damper algorithm for an edge start region (Section 4.3.3). The method returns the union of three visibility computations, two from the endpoints of $\bar{e}$, and one from the interior. The endpoint computations are performed by convex visibility analysis. The reachable volume for each computation is a cylinder of radius ϵ_t, which extends from the endpoint to the commanded position, similar to Figure 3.13. The interior computation is performed by y-convex visibility analysis, using the reachable volume of Figure 4.23.

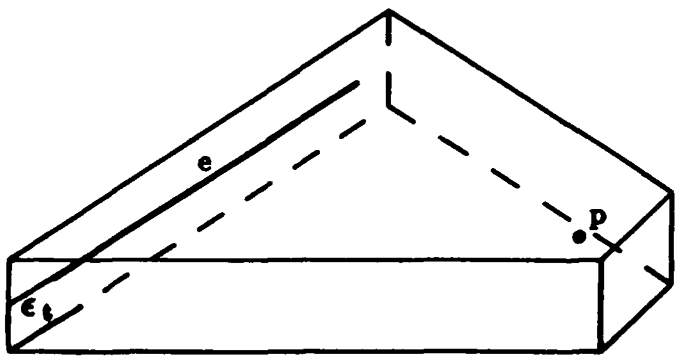


Figure 4.23: All possible free space trajectories from a start edge $\bar{e}$ under generalized spring dynamics are contained in the volume shown here.

4.4.3 Reachable Points From a Polygon

Now suppose that the robot is located in a polygon S, and is assigned a commanded position $\mathbf{p}$. Figure 4.24 shows the volume of free space trajectories that can result. This volume contains all possible commanded trajectories, and all points that are within the trajectory error ϵ_t of the commanded trajectories.

Straight-line reachable points from S can be computed by the same method used in the generalized damper algorithm for a polygonal start region (Section 4.3.4). We begin by computing straight-line reachable points from the bounding edges of S, using the methods described above. Straight-line reachable points from the interior of S can then be computed by orthographic visibility analysis, using the reachable volume of Figure 4.24. The union of all of these reachable points is returned.

4.4.4 Sliding and Sticking

When a robot modeled as a generalized spring strikes a face F at a point $\mathbf{s}$, it slides along the surface of F within the projection of the free space cylinder onto the surface, towards the projection of the commanded position onto the face (Figure 3.14). It stops when its spring force is equaled by a reaction force of F. The subset of F in which sticking is possible with respect to $\mathbf{p}$ under position sensing and velocity uncertainty is shown in Figure 3.16. This region is called the *weak sticking region* of F with respect to $\mathbf{p}$. Also shown in the figure is the subset of F in which sticking is guaranteed. This region is called the *strong sticking region* of F with respect to $\mathbf{p}$. No sliding is possible in this region.

Here is a proposed algorithm to simulate sliding on a face F from a start polygon S under a commanded position $\mathbf{p}$:

Algorithm 4.4 *Sliding Simulation (Generalized Spring)*

```
1. Subtract the strong sticking region of $F$ from $S$.
2. Compute the cylinder $R \subseteq F$ which contains the sliding trajectories from $S$.
```

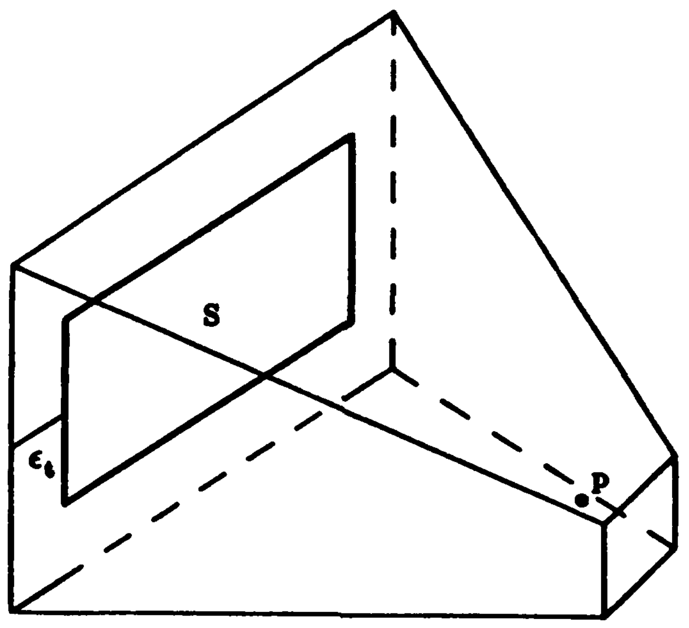


Figure 4.24: All possible free space trajectories from a start polygon S under generalized spring dynamics are contained in the volume shown here.

3. *Subtract the strong sticking region of F from R.*
4. *Return R.*

4.5 Sensing Termination

This section describes the simulation of a motion m under sensing termination. Sensing termination is specified by the parameters P_s and F_s. The robot is to stop if the sensed position is in the set P_s, and the direction of the sensed force is in the set F_s.

Definition 4.1 *A contact point* x *is* **weakly consistent** *with a set of position measurements P_s and a set of force measurements F_s if* x *is within ϵ_s of some point in P_s, and some vector in the friction cone of* x *is within ϵ_f of a vector in F_s.*

Definition 4.2 *A contact point* x *is* **strongly consistent** *with a set of position measurements P_s and a set of force measurements F_s if $Ball(\mathbf{x}, \epsilon_s)$ is contained in P_s, and the friction cone of* x*, expanded in angle by θ_f, is contained in F_s.*

Let T_w be the set of contact points which are weakly consistent with P_s and F_s. The points in T_w may stop the robot if reached. Let T_s be the set of contact points which are strongly consistent with P_s and F_s. The points in T_s are guaranteed to stop the robot if reached.

To simulate m, we make the following modifications to the simulation framework (Algorithm 4.1):

- We must treat T_s as a strong sticking region during sliding, for sliding is not possible from the points of T_s. In Algorithms 4.3 and 4.4, we must subtract T_s from the start region S. After computing the sliding region R, we must then subtract T_s from R.
- We must treat T_w as a set of weak sticking points in step 4 of Algorithm 4.1, to account for possible sensing termination.

Chapter 5

Teaching

This chapter describes a new method for teaching a compliant motion strategy to a robot. The input to the teaching system consists of:

- a polyhedral environment, representing obstacles in the configuration space of a Cartesian robot which can translate in three dimensions. We will assume that the environment is bounded, and that the faces of the polyhedra are convex.
- a start region, given by a list of convex surface polygons in the environment.
- a goal region, also given by list of convex surface polygons in the environment.

This is the same as the input to the verifier (Chapter 4), except that the verifier was given a compliant motion, and asked to check its correctness. The teaching system is to synthesize a compliant motion strategy. Figure 1.6 shows an example of a compliant motion strategy.

Section 5.1 presents an overview of the teaching system. Section 5.2 presents an example of the implemented teaching system in operation. Sections 5.3 and 5.4 describe the algorithms used by the teaching system.

5.1 Overview of the Teaching System

Figure 5.1 shows an operational view of the teaching system. The teacher submits a problem to the system, consisting of an environment, a start region, and a goal region. The programming system displays the start and goal regions graphically, and prompts the teacher. The teacher then submits a commanded motion m, which can be either a commanded position or velocity. In principle, the commanded motion could be entered by guiding the robot, or with a light pen. For our initial implementation, the commanded motion was simply typed in. The commanded motion should be one which the teacher hopes will reach the goal region reliably from the start region. If this is impossible, then the teacher should submit what is hoped will be the last motion in a reliable sequence of motions. Minimally, the motion should be one which reliably reaches the goal region from *somewhere*.

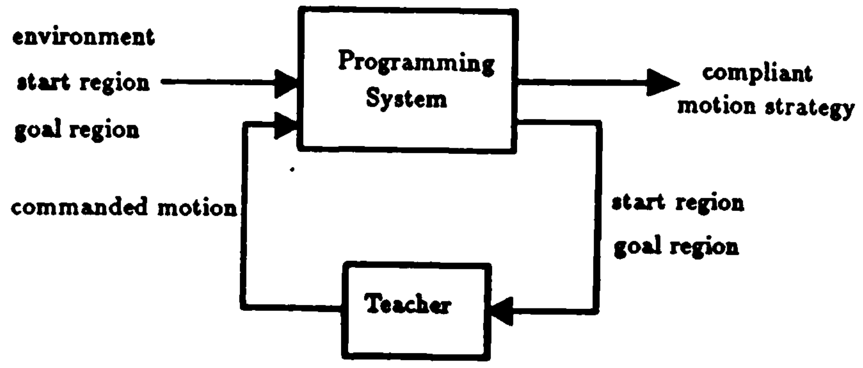


Figure 5.1: An operational view of the robot teaching system. The teacher submits a problem to the system, consisting of an environment, a start region, and a goal region. Then, the teacher is asked to supply suggested motions, which can be either commanded positions or velocities.

The system then computes a set R of contact configurations from which the goal region can be reached reliably via m. R is stored in a table, along with m and the subset of the goal region that can be reached from R. R is now said to be *solved*, and is added to the goal region. If a subset S' of the start region is recognizably contained in R despite sensing uncertainty, then S' is solved, and can be removed from the start region. If the new start region is empty, then the problem is solved. Otherwise, the system displays the new start and goal regions, and user interaction continues.

By iteratively reducing the size of the start region, and increasing the size of the goal region, it is hoped that the user and system can together converge on a successful strategy. However, this is not guaranteed. For instance, if no successful strategy exists under the given uncertainty bounds, then the iterative process will fail.

If the teacher successfully solves the entire start region, then the table of solved regions is used to generate a directed, acyclic graph, representing the resulting compliant motion strategy. Each solved region R in the table is accompanied by (a) a commanded motion m which is to be issued upon reaching R, and (b) the solved regions that m might reach from R. From each solved region R, we generate a node in the strategy graph. An arc is attached to the node, labeled by the motion m. The heads of the arc point at the nodes corresponding to the solved regions that m might reach from R. Following each motion on execution, the robot uses sensors to determine which solved region it is in. If the solved region is a goal region, then the robot stops. Otherwise, it executes the next commanded motion.

This teaching system has the following advantages:

1. If a proposed motion fails, it is not scrapped. The system attempts to use the motion as a building block for a reliable procedure.

2. Conditional tests, which determine the solved region that each motion termi-

nates in, are inferred automatically by the programming system. Conditional tests make a motion strategy reliable under sensing and control uncertainty, but they are difficult for a teacher to specify.

3. Motion termination conditions are inferred automatically by the programming system. Termination conditions are also difficult for a teacher to specify.

4. The teacher need not test the resulting motion strategy; the system ensures its reliability.

To implement the teaching system, we must answer the following questions:

- *How can we determine whether a start point* s *is recognizably contained in a solved region R despite sensing uncertainty?*

For s to be recognizably contained in R, it must not be confusable with any contact point outside of R by position and force sensing, under the given sensing uncertainties. Intuitively, two points **x** and **y** are confusable of they are capable of generating the same position and force measurements. Section 5.3 will develop this concept in more detail.

- *How can we compute the solved region R for a given goal region G and commanded motion m?*

Here is an overview of the computation. Further details are given in Section 5.4.

R is called a *pre-image* of G under m [Lozano-Pérez, Mason, and Taylor 1984]. To compute R, we will extend the algorithm of Erdmann [1984,1986], under both the generalized damper and spring trajectory models. In Erdmann's algorithm, a pre-image of G under m is computed by geometric backprojection from a subset X of G. X is called a *backprojection base*. One way to pick X is to choose the set of recognizably contained points in G. This ensures that when the robot reaches X, it can recognize that it has attained G.

Geometric backprojection from X under m computes contact points from which the robot is guaranteed to reach X under m despite sensing and control uncertainty. This type of backprojection region is called a *strong backprojection region*. Another type of backprojection region, a *weak backprojection region*, contains contact points from which the robot *might* reach X under sensing and control uncertainty.

Trajectories originating in the strong backprojection region of X must avoid contact points outside of X where the robot might terminate. For example, the robot must avoid a weak sticking point under m that is outside of X. With this in mind, the strong backprojection region of X under m can be computed as the complement of the weak backprojection region of B under m, where B is the set of points outside of X where the robot might terminate.

We have thus reduced the problem to that of computing the weak backprojection region of a set B under motion m. An algorithm is presented to do this in Section 5.4.

5.2 An Example

Figure 5.2 shows an environment consisting of a square hole in an obstacle surface. Figure 5.3 shows another view of the hole. This configuration space environment corresponds to a large number of mechanical insertions using three-dimensional translational motion. The simplest of them is the insertion of a rectangular peg into a rectangular hole. The peg is assumed to be rotationally aligned with the hole at the start. The width of the hole is 2 inches. This corresponds to a peg clearance of 1 inch. The coefficient of friction of the hole surfaces is .25.

The start region is an edge above the hole, as shown in Figures 5.2 and 5.3. This start region corresponds to a contact configuration in which the peg is in contact with a surface above the real hole. This start region implies that the peg is already laterally aligned with the hole.

The goal region is the bottom of the configuration space hole. This region corresponds to the peg when it is fully inserted into a real hole.

We will use generalized spring motions in our solution strategy. We will assume the following sensing and control error bounds:

- $\epsilon_s = .2$ inches
- $\theta_f = .3$ radians
- $\epsilon_p = .2$ inches
- $\theta_v = .25$ radians

Under these error bounds, the maximum free space trajectory error ϵ_t given by Equation 3.21 is equal to .218 inches. This is approximately one quarter of the peg clearance.

The teaching system initiates the session by prompting the teacher for a commanded position. The teacher is to submit the last commanded position in the strategy. Since the goal region is the bottom of the hole, it makes sense to command a position $\mathbf{p}_1$ which is beneath the hole. $\mathbf{p}_1$ should be at least .2 inches beneath the hole, so that the robot does not stop in free space. Figure 5.4 shows $\mathbf{p}_1$.

The programming system must now compute a backprojection base. It turns out that every point in the goal region is recognizably contained in the goal by force sensing under the given error bound, since the bottom of the hole is perpendicular to the sides of the hole. Thus, the backprojection base is equal to the entire goal.

The robot must avoid sticking in a nongoal point on the way to $\mathbf{p}_1$. The programming system thus computes the nongoal weak sticking points under $\mathbf{p}_1$. These points are shown in Figure 5.4. The weak backprojection region of these points under $\mathbf{p}_1$ is shown in Figure 5.5. These weak backprojection points are not reliable starting points under $\mathbf{p}_1$, for from them the robot may stick in nongoal points on the way to $\mathbf{p}_1$. The complement of these points comprises the strong backprojection region of the goal region under $\mathbf{p}_1$. These strong backprojection points are shown in Figure 5.6. As can be seen from the figure, the strong backprojection region consists of the bottom and sides of the hole. The sides of the hole are new solved regions. Since the start region does not intersect these new solved regions, the start region is not solved yet, so user interaction must continue.

Since the sides of the hole are now solved, and the start region is an edge to the left of the hole, the teacher picks the next commanded position $\mathbf{p}_2$ behind the right side of the hole. The depth of $\mathbf{p}_2$ was chosen to be halfway into the hole, so that the robot will drop into the hole after sliding to it. It is important not to make $\mathbf{p}_2$ too deep, or else the robot may stick on the way to the hole. Figure 5.7 shows $\mathbf{p}_2$.

The programming system must now compute a backprojection base for $\mathbf{p}_2$. Once again, every point in the solved regions is recognizably contained in the goal by force sensing. Thus, the backprojection base is equal to the entire set of solved regions.

The robot must avoid sticking in an unsolved point on the way to $\mathbf{p}_2$. The programming system thus computes the unsolved weak sticking points under $\mathbf{p}_2$. These points are shown in Figure 5.7. The weak backprojection region of these points under $\mathbf{p}_2$ is shown in Figure 5.8. These weak backprojection points are not reliable starting points under $\mathbf{p}_2$, for from them the robot may stick in an unsolved point on the way to $\mathbf{p}_2$. The complement of these points comprises the strong backprojection region of the solved regions under $\mathbf{p}_2$. The strong backprojection points are shown in Figure 5.9. These points are now solved. As can be seen from the figure, the new solved regions contain the start region. We must be careful that points in the start region can recognize that they are solved points. The programming system thus computes the recognizably contained subset of the new solved regions. The recognizable subset is shown in Figure 5.10. The start region is fully contained in this region, and thus is completely solved. The resulting compliant motion strategy consists of the decision-free sequence of motions $(\mathbf{p}_2, \mathbf{p}_1)$.

5.3 Recognizable Containment

Once we have computed a solved region R, we can eliminate any points from the start region S which are recognizably contained in R despite sensing uncertainty. The following definitions will aid us in this task:

Definition 5.1 *Two contact points* $\mathbf{x}$ *and* $\mathbf{y}$ *are* **confusable** *by position and force sensing if both of the following conditions hold:*

1. $\mathbf{x}$ *and* $\mathbf{y}$ *are within* $2\epsilon_s$ *of each other.*
2. *The friction cones of* $\mathbf{x}$ *and* $\mathbf{y}$ *are within an angle of* $2\theta_f$ *of each other.*

Intuitively, $\mathbf{x}$ and $\mathbf{y}$ are confusable of they are capable of generating the same position and force measurements.

Definition 5.2 *The* **confusable set** *of a contact point* $\mathbf{x}$ *is the set of all contact points that are confusable with* $\mathbf{x}$.

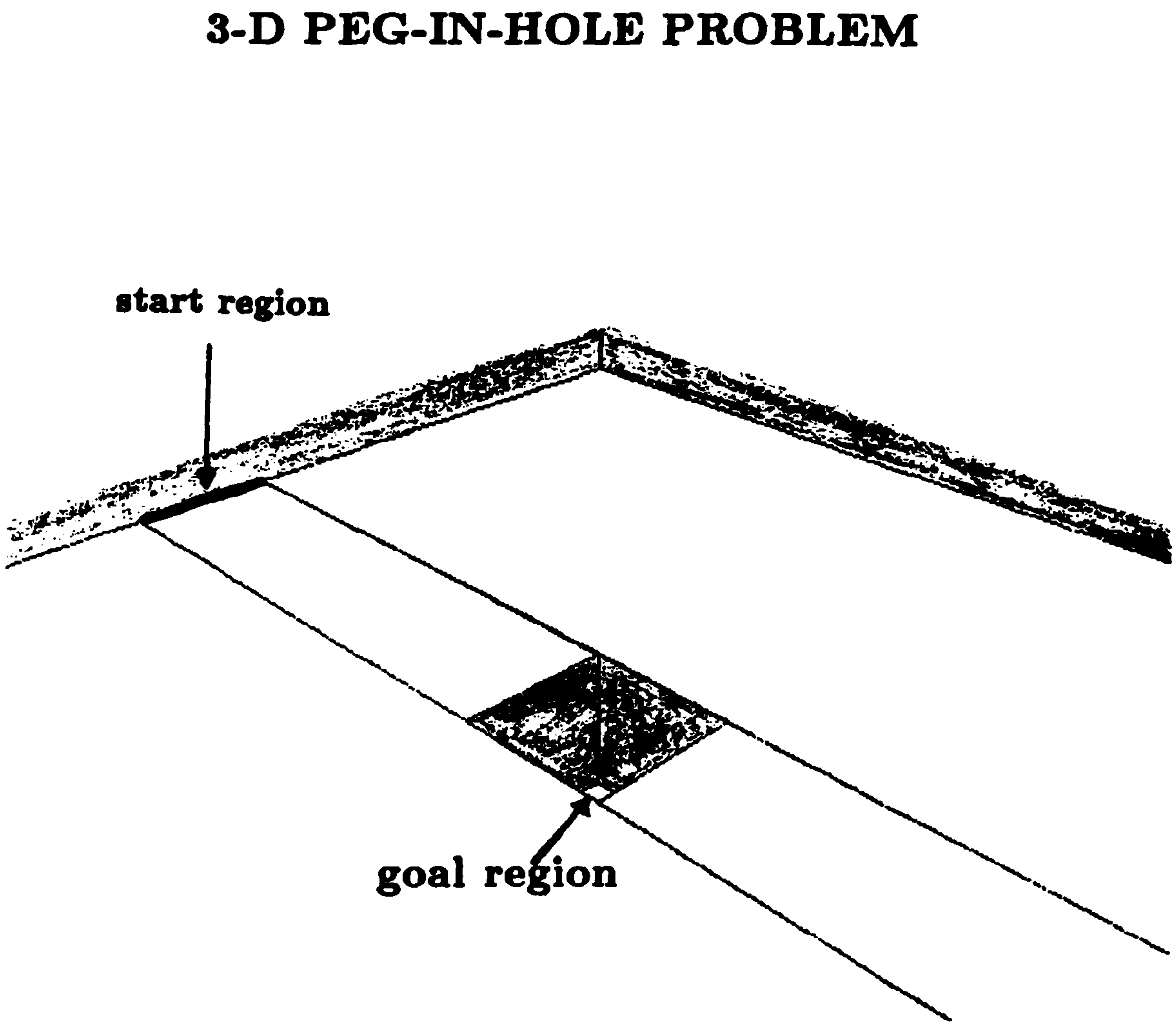


Figure 5.2: An environment consisting of a square hole in an obstacle surface. The start region is an edge on the surface of the obstacle. The goal region is the bottom of the hole.

Hole Problem, Top View

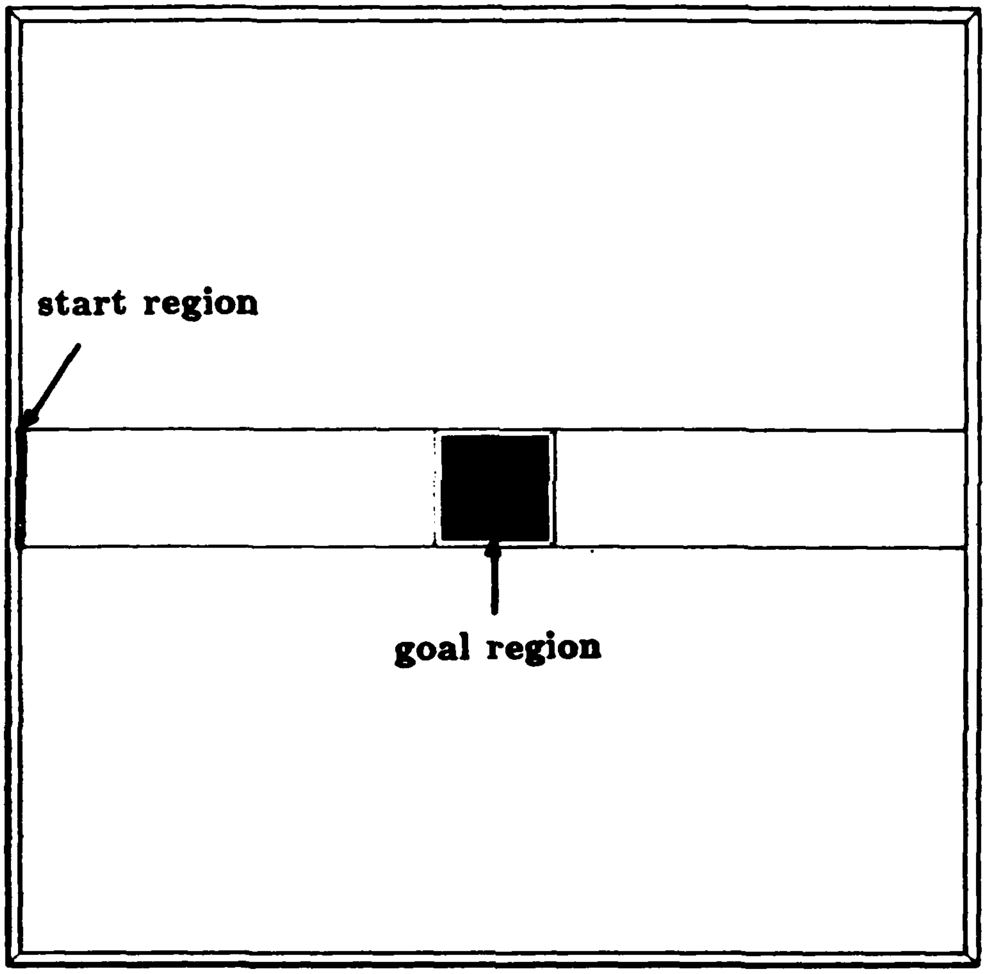


Figure 5.3: A top view of the hole environment. The start region and goal regions are shown.

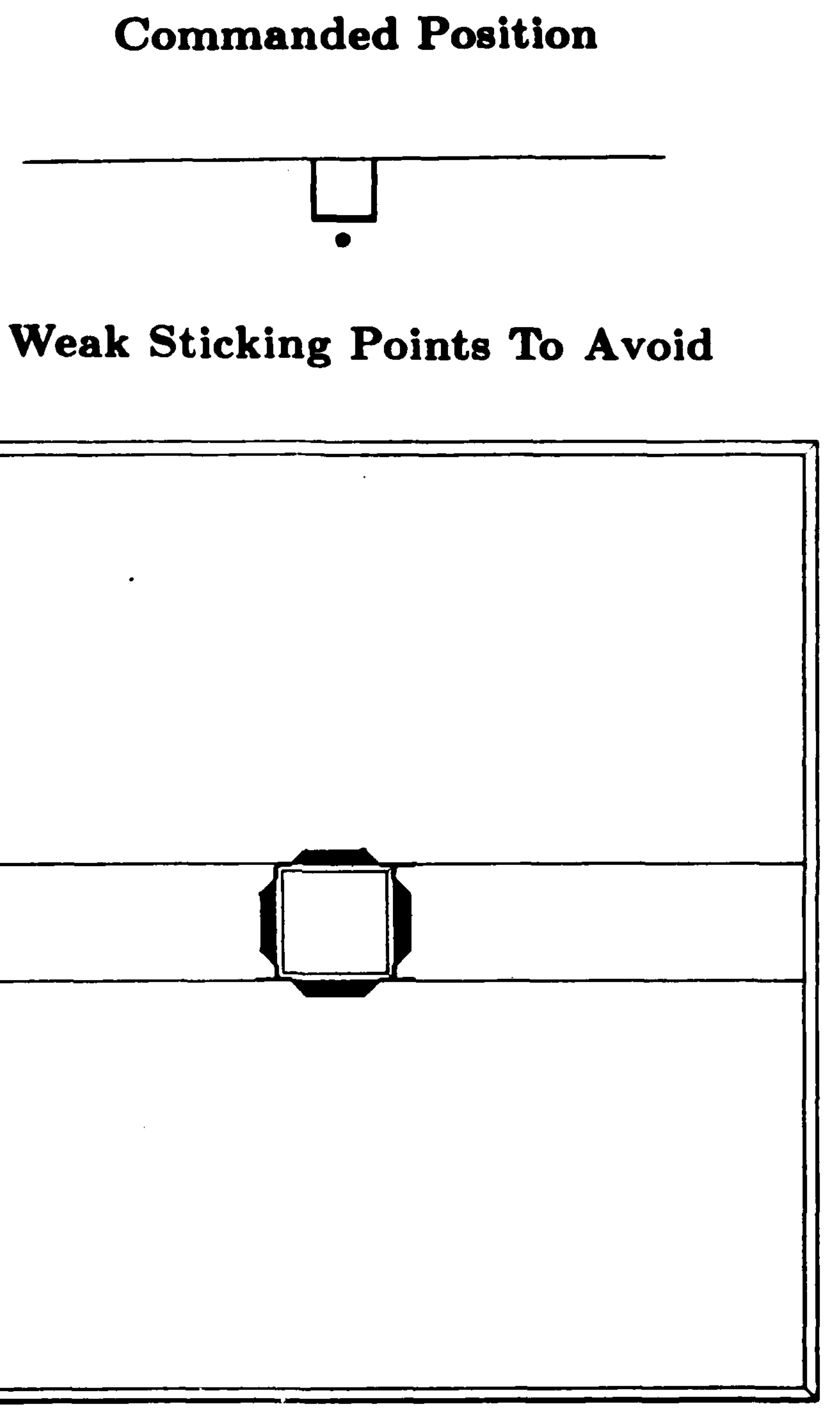


Figure 5.4: The first commanded position $\mathbf{p}_1$ is beneath the hole, as shown at top. The nongoal weak sticking points under $\mathbf{p}_1$ are shown at bottom.

Weak Backprojection of Sticking Regions

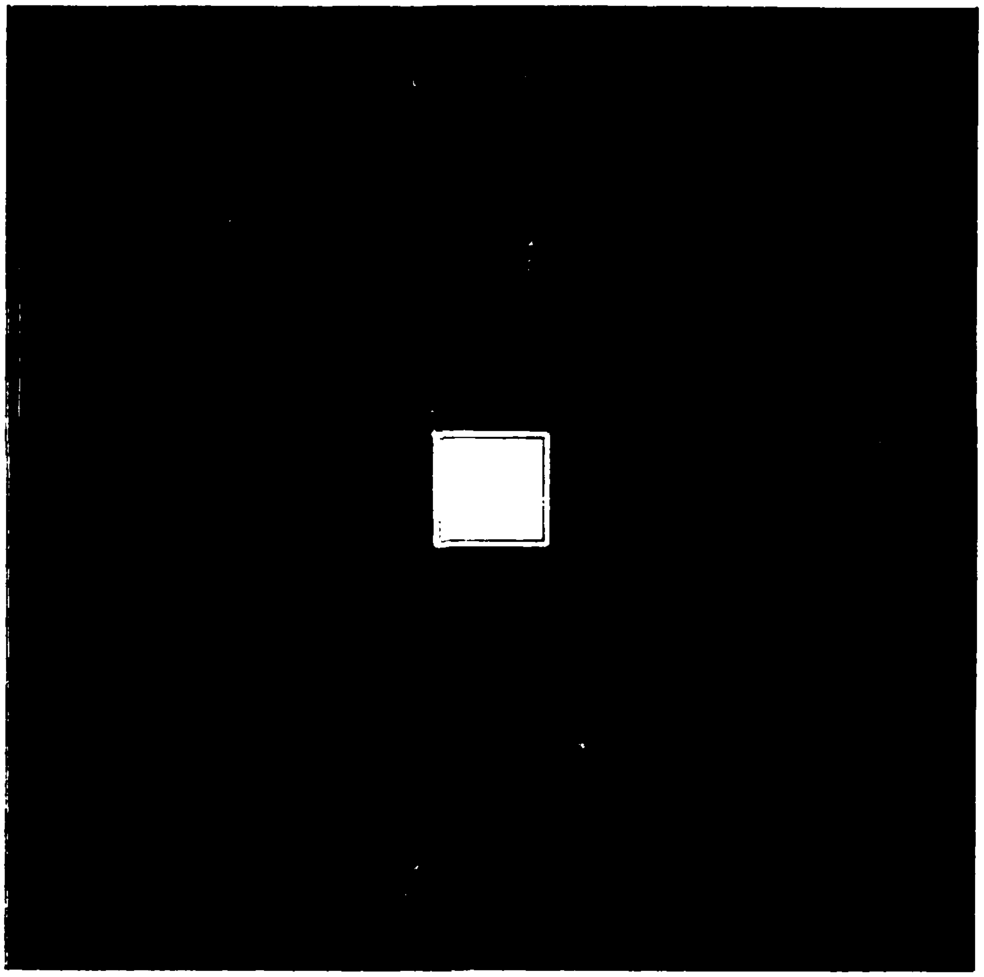


Figure 5.5: The shaded regions comprise the weak backprojection region of the nongoal weak sticking points under $\mathbf{p}_1$.

Strong Backprojection of Goal Region

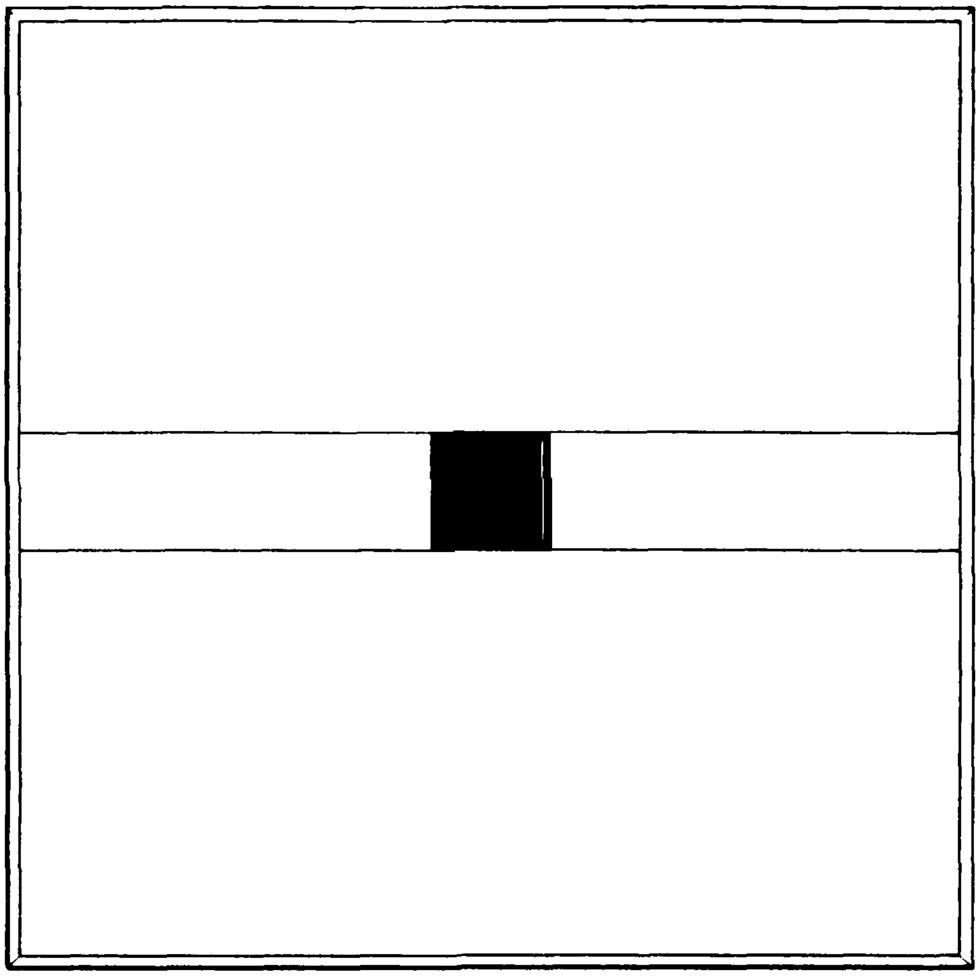
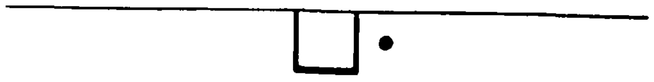


Figure 5.6: The shaded regions comprise the strong backprojection region of the hole bottom under $\mathbf{p}_1$.

Commanded Position



Weak Sticking Points To Avoid

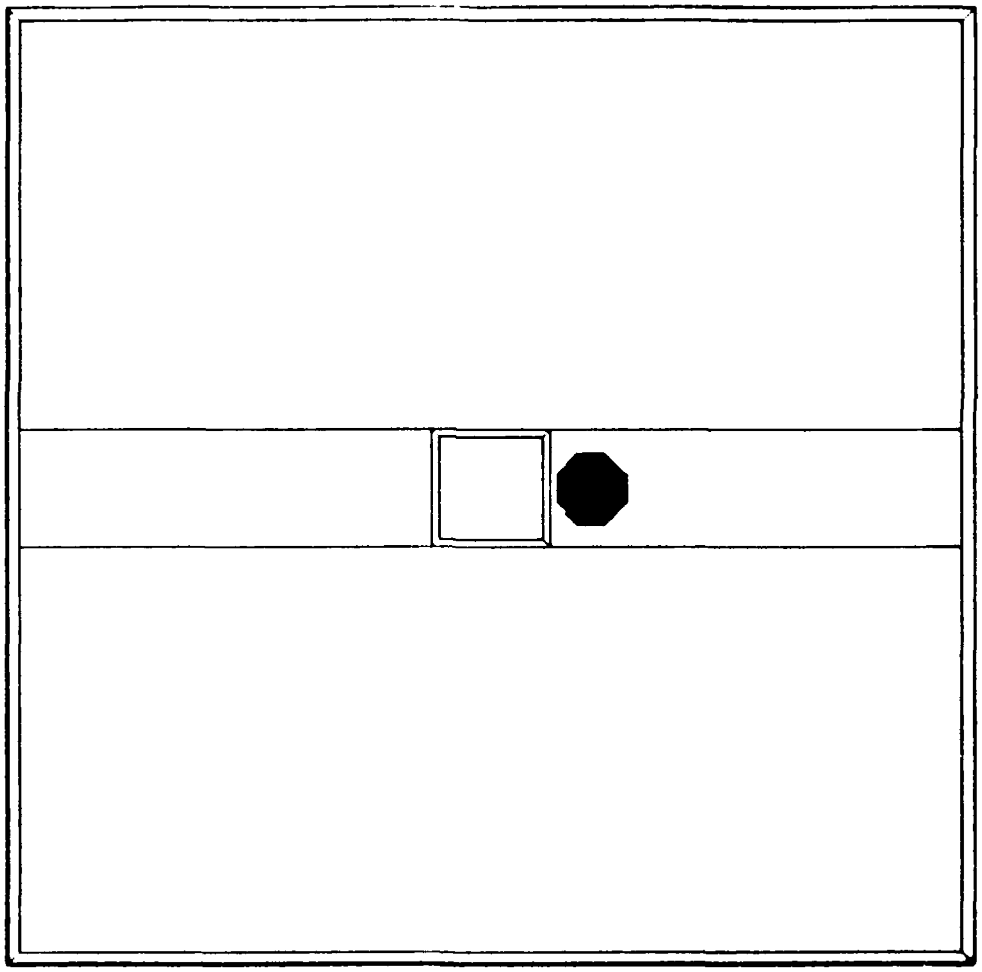
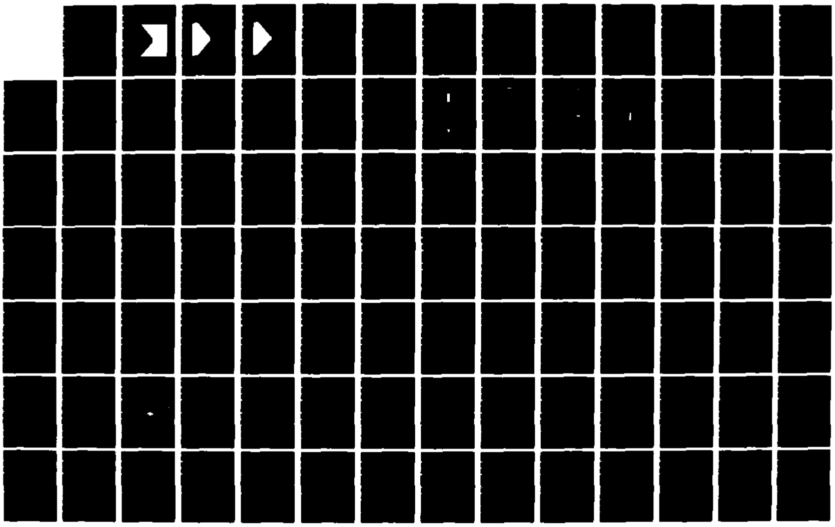
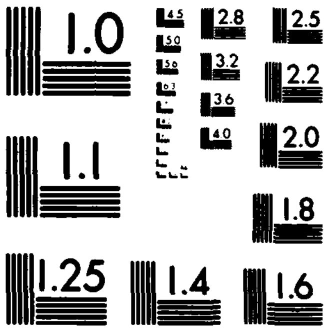


Figure 5.7: The second commanded position $\mathbf{p}_2$ is chosen to the right of the hole. as shown at top. The nongoal weak sticking points under $\mathbf{p}_2$ are shown at bottom

AD-A186 418 PLANNING AND TEACHING COMPLIANT MOTION STRATEGIES(U) 2/3
MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL
INTELLIGENCE LAB S J BUCKLEY JAN 87 AI-TR-936
UNCLASSIFIED N00014-85-K-0124 F/G 12/9 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3

Weak Backprojection of Sticking Regions

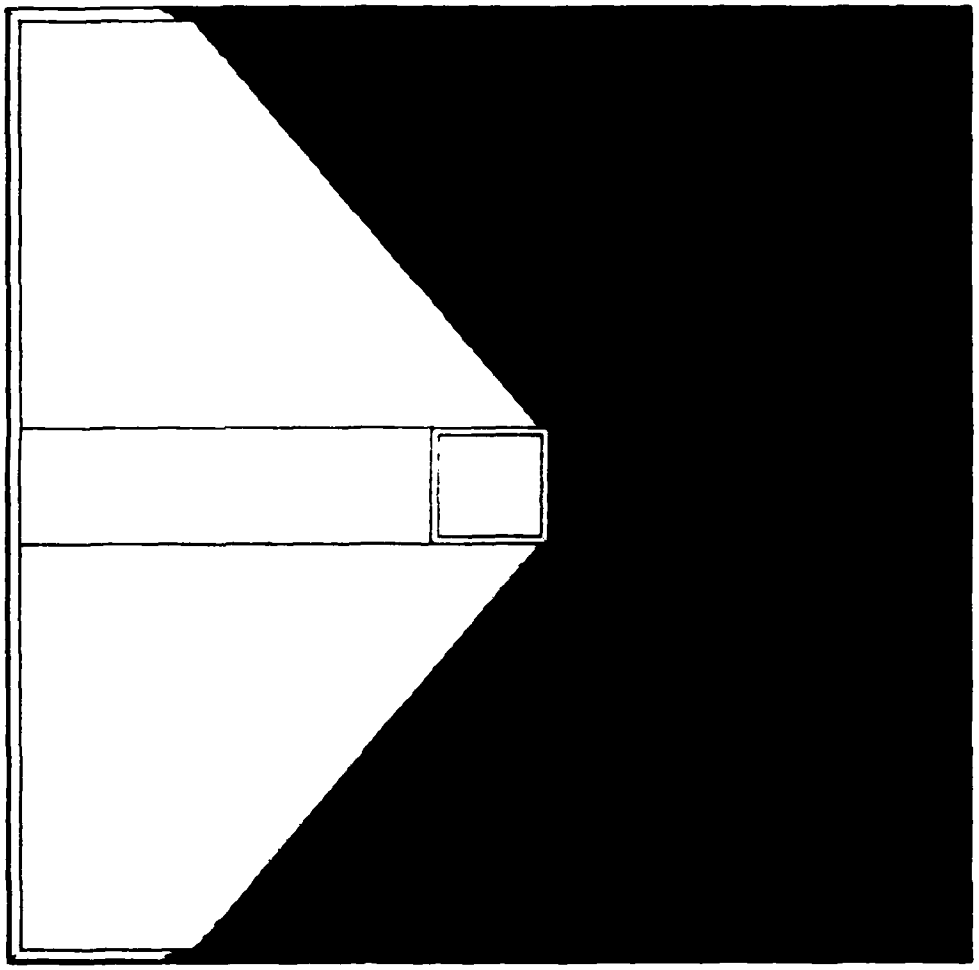


Figure 5.8: The shaded regions comprise the weak backprojection region of the unsolved weak sticking points under $\mathbf{p}_2$.

Strong Backprojection of Goal Region

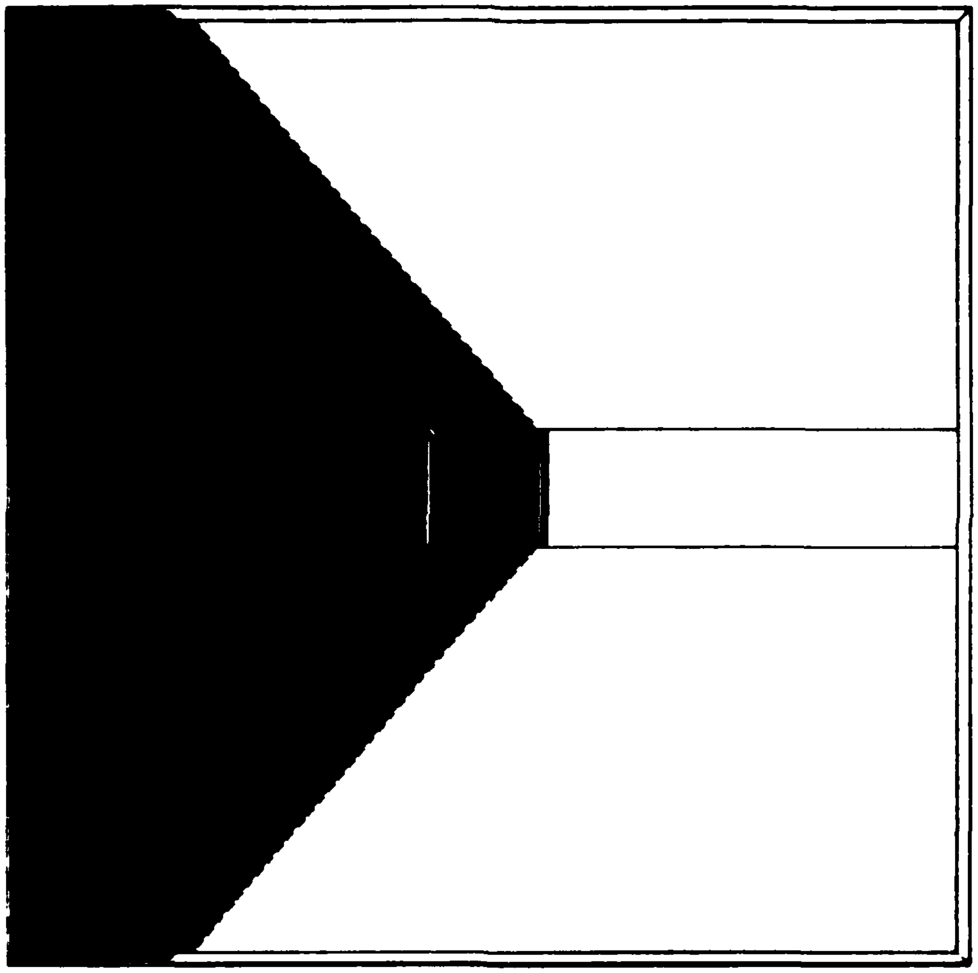


Figure 5.9: The shaded regions comprise the strong backprojection region of the solved regions under $\mathbf{p}_2$.

Recognizable Subset of Strong Backprojection

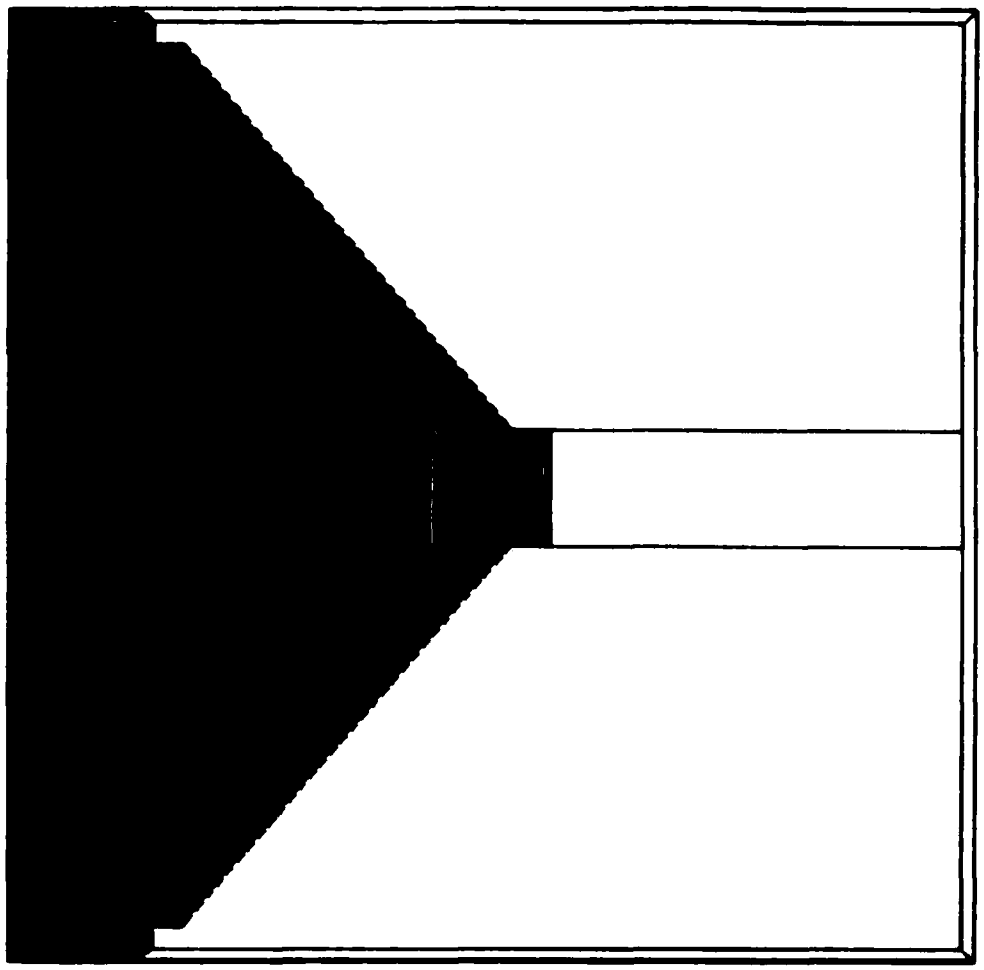


Figure 5.10: The shaded regions comprise the recognizably contained subset of the shaded regions of Figure 5.9.

Definition 5.3 *Let R be a set of contact points, and let* s *be a contact point.* s *is* **locally weakly consistent** *with R if a point in the confusable set of* s *is contained in R.*

Definition 5.4 *Let R be a set of contact points, and let* s *be a contact point.* s *is* **locally strongly consistent** *with R if the confusable set of* s *is contained in R.*

If a contact point s is locally strongly consistent with R, then s is recognizably contained in R under local sensing. We use the qualifier *local* here because in some cases there are global constraints on the configuration of a robot that can be used to establish containment. For example, suppose that the robot is known to have made a motion m from a start region A, prior to terminating at s. Then the robot is constrained to lie in the simulation $T_m(A)$ (Chapter 4). In this case, if the intersection of the confusable set of s with $T_m(A)$ is contained in R, then s is recognizably contained in R.

The following algorithm computes whether s is locally strongly consistent with R:

Algorithm 5.1 *Local Strong Consistency*

1. *Compute the set R_c of contact points in $\overline{R}$ (the complement of R) that are within $2\epsilon_s$ of R, and whose friction cones are within $2\theta_f$ of R's friction cone.*
2. *Grow R_c by $2\epsilon_s$ in three dimensions.*
3. *Clip the resulting volume out of R.*
4. *Test whether* s *is in the reduced set R.*

This algorithm eliminates any points from R which are confusable with other contact points, leaving only points which are locally strongly consistent with R. Note that step 1 is made easier by the fact that all points in a face, edge, or vertex have the same friction cone.

5.4 Pre-Images

The central task of the teaching system is to compute a pre-image region R from which a taught motion m can reliably terminate in the goal region G. Pre-images were first proposed by Lozano-Pérez, Mason, and Taylor [1984] as a subtask in a backward chaining motion planner. Their proposal did not specify an implementation for pre-images. Erdmann [1984,1986] showed that for certain classes of termination conditions, pre-images can be computed by geometric backprojection from a backprojection base. The idea is to pick a termination condition, such as local sensing, and reduce the goal region to a backprojection base X which is recognizably contained in the goal region under the termination condition and its associated sensing uncertainty. Backprojection from X computes points from which the robot is guaranteed to reach X despite sensing and control uncertainty. When the robot reaches a point $\mathbf{x} \in X$ from a point in the backprojection region, it can then recognize $\mathbf{x}$ as strongly consistent with G under the termination condition and its associated sensing uncertainty. Erdmann implemented his scheme for planar robots with rotation, under the generalized damper trajectory model. In this section, we will extend Erdmann's algorithm to three Euclidean dimensions, under both the generalized damper and spring trajectory models, for the case of motions which start and stop in contact space. We will assume the following termination condition:

- The robot may terminate by sticking at a goal point. (This does not have to be specified in commanded motions.)
- The robot may terminate by sensing a goal point which is strongly consistent with G under local sensing, with respect to the forward projection $F_m(R)$ of the start region R under the commanded motion m.

Erdmann called this the *termination predicate without history or time*. The role of the forward projection $F_m(R)$ in termination is discussed below. Most of the ideas that follow are applications or interpretations of Erdmann's thesis.

5.4.1 The Backprojection Base

Our termination condition defines two types of reliable termination points:

1. Goal points at which the robot is guaranteed to stick.
2. Goal points that can be consistently recognized as goal points by sensing because their confusable set is contained in $G \bigcup \overline{F_m(R)}$. Such points are said to be *strongly consistent with G under the forward projection $F_m(R)$*.

The set of reliable termination points comprises the backprojection base X. The first type of base point is a goal point at which the robot is guaranteed to stick. The second type of base point is a goal point which can only be confused with points in G or with unreachable points in $\overline{F_m(R)}$. A base point whose confusable set is a subset of G can be computed by Algorithm 5.1. A base point with confusable points in $\overline{F_m(R)} - G$ can be computed by substituting $G \bigcup \overline{F_m(R)}$ for G in Algorithm 5.1, and intersecting the result with G. Regrettably, this is a circular definition, for R is precisely what we are trying to compute. It is currently an unsolved problem to remove this circularity without loss of pre-image points. Here, we propose two algorithms to remove the circularity such that approximate pre-images can be computed:

- **Backprojection Base Algorithm A**

  One way to remove the dependency on R is to choose an initial approximation set R', and compute X with respect to $F_m(R')$. For correctness, it is necessary that R' contain R, so that we do not leave any points in X which are confusable

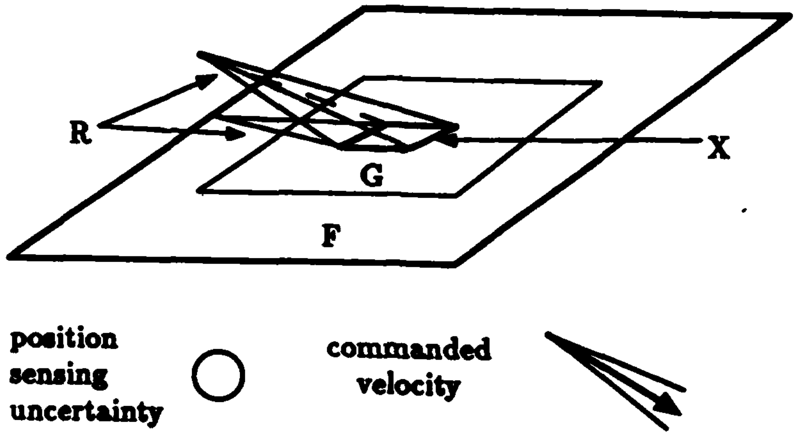


Figure 5.11: The goal region is the surface polygon G, on the face F. The position sensing and velocity uncertainties are shown. A commanded velocity $\mathbf{v}$ is given which slides on F. The backprojection base X is chosen as the set of goal points which are locally strongly consistent with G. The pre-image R that can be computed by backprojecting from X consists of a sliding region and a free space region, as shown.

with nongoal points in $F_m(R - R')$. In practice, however, it is not clear how one would choose R'. Choosing R' as the entire configuration space would result in a forward projection that is also equal to the entire configuration space. X would then consist only of sticking goal points and locally strongly consistent goal points. Choosing R' as the entire configuration space works well for the case where sliding to the goal is desired, as illustrated in Figure 5.11. In general, if $R' \neq R$, then the technique might eliminate points from the backprojection base whose nongoal interpretations would be unreachable from R, as illustrated in Figure 5.12.

- **Backprojection Base Algorithm B**

  Another way to remove the dependency on R is to pick the backprojection base X independently. There are two modifications to Erdmann's algorithm that make this possible. First, the termination condition is modified to stop when the sensors are locally weakly consistent with G, rather than locally strongly consistent. Second, the backprojection region is computed so as to avoid nongoal points which are confusable with goal points. The second modification is necessitated by the change in the termination condition. One choice for the backprojection base X is the entire goal region G. This technique may overconstrain the forward projection so as not to contain some pre-image points. For example, the technique cannot compute sliding motions, since such motions must pass through a confusable nongoal layer surrounding the goal, as illustrated in Figure 5.13. The technique does work well for surface goals that are reachable by free space trajectories, also illustrated in Figure

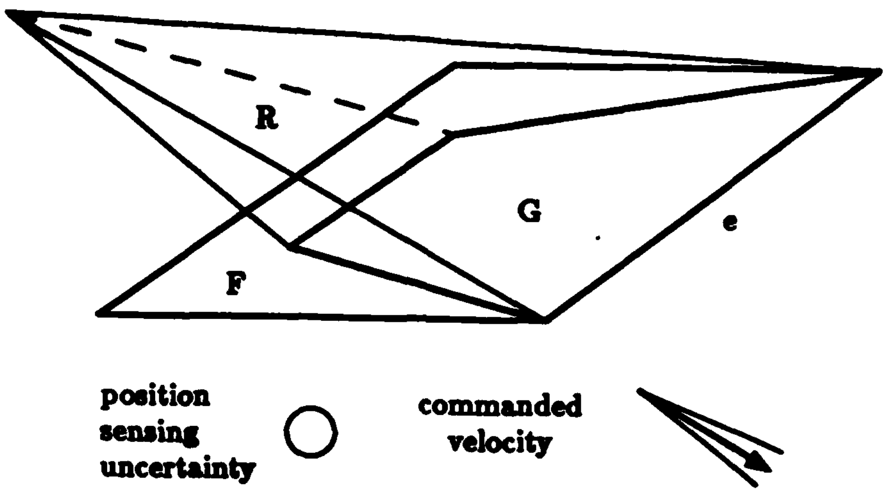


Figure 5.12: The goal region is the union of the surface polygon G and the edge $\bar{e}$, on the face F. The position sensing and velocity uncertainties are shown. A commanded velocity is given which slides on F, and sticks on $\bar{e}$. The free space region R is a pre-image, but R cannot be computed by Backprojection Base Algorithm A when $R' \neq R$, since some of the backprojection base points would be confusable with nongoal points in $F_m(R')$.

5.13.

We will settle for a hybrid solution. First, using Backprojection Base Algorithm A, we will backproject from a base of points which are locally strongly consistent with G. This will compute pre-image points from which G can be reached by sliding, among others. Then, using Backprojection Base Algorithm B, we will backproject from the entire goal region, avoiding nongoal points that are confusable with goal points. This will compute pre-image points from which the goal can be reached by free space trajectories. We will return the union of all of the backprojection points.

5.4.2 Backprojection

Backprojection from a set X under a motion m is basically a reversal of the simulation presented in Chapter 4. In the simulation, we were give a start region R and a commanded motion m, and asked to compute the set X of possible termination points. Here, we are given m and X, and asked to compute R.

There are actually two types of backprojection regions. A weak backprojection region $W_m(X)$ of a set X under a commanded motion m is a set R of contact points from which m might reach X. A strong backprojection region $S_m(X)$ of a set X under a commanded motion m is a set R of contact points from which m is guaranteed to reach X. Here, we need to compute the strong backprojection region of a backprojection base X, since our ultimate goal is to compute reliable motions.

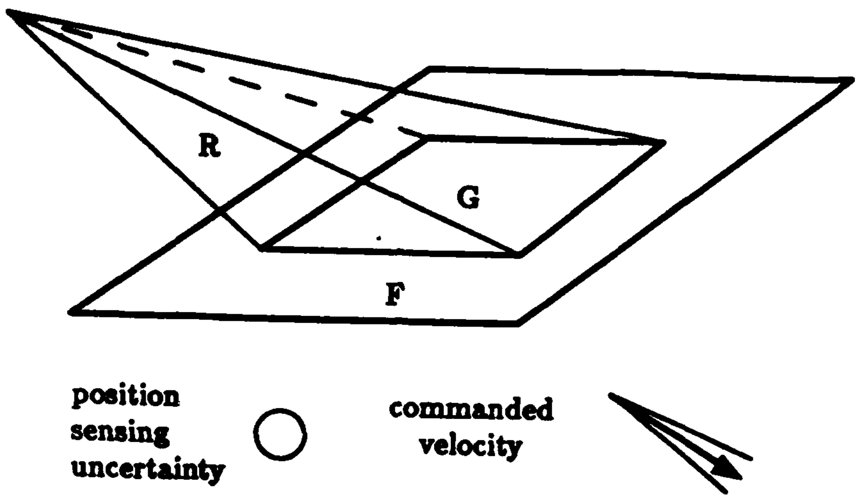


Figure 5.13: This is the same example as Figure 5.11. Here, the backprojection base is chosen as the entire goal region. Sensing termination is by local weak consistency. Backprojection from G must avoid all nongoal points which are confusable with goal points. Since G is surrounded by confusable points, sliding to G is impossible. However, the backprojection region in free space is much larger than in Figure 5.11.

$S_m(X)$ can be formulated in terms of $W_m(X)$. Trajectories originating in $S_m(X)$ must avoid nongoal points where the robot might terminate. These include (a) weak sticking points under m, and (b) nongoal points that are confusable with goal points, if we are using Backprojection Base Algorithm B. The latter type of point must be avoided because Backprojection Base Algorithm B uses a weak consistency termination condition. With this in mind, $S_m(X)$ is equal to the complement of $W_m(B)$, where B is the set of nongoal points where the robot might terminate. Avoiding nongoal termination points ensures that the robot will eventually reach X, since the environment is bounded.

We have thus reduced the problem to that of computing $W_m(B)$. An algorithm for weak backprojection is now presented.

Algorithm 5.2 *Weak Backprojection*

1. *Initialize W to B.*
2. *Add to W the set of contact points from which the robot might reach B via sliding under m. These new points are called* **weak backsliding points** *from B under m.*
3. *Compute the set Y of contact points from which the robot might reach W via straight-line free space trajectories under m. Add Y to W.*
4. *Recursively call the algorithm with B equal to Y, minus any points that have already been backprojected from in previous recursive calls. Add to W any new backprojection points that are returned.*

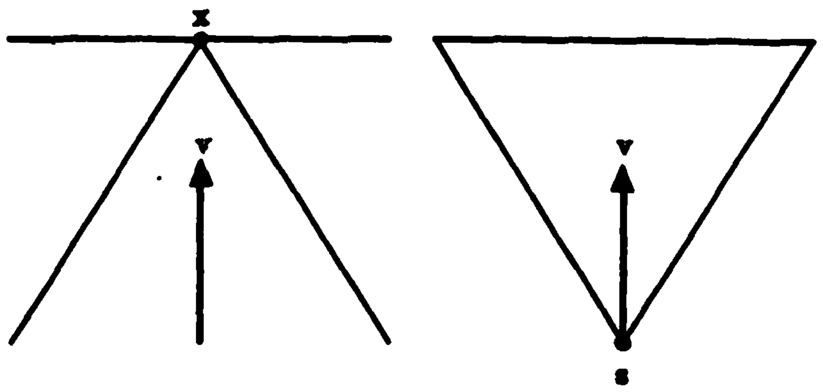


Figure 5.14: The left side shows the cone which contains straight-line weak backprojection points from a point x. The right side shows the cone which contains straight-line reachable points from a start point s. The cones are exactly the same.

5. *Return W.*

We are left with two subtasks, the computation of straight-line weak backprojection points, and the computation of weak backsliding points. The next two subsections discuss these computations for the generalized damper and spring trajectory models, respectively.

5.4.3 Generalized Damper Weak Backprojection

Straight-line weak backprojection for the generalized damper trajectory model is equivalent to computing straight-line reachability (Section 4.3), with the start and goal regions interchanged. Consider Figure 5.14. The left side of the figure shows the cone which contains straight-line weak backprojection points from a point x. The right side shows the cone which contains straight-line reachable points from a start point s. The cones are exactly the same. This makes sense, since the straight-line weak backprojection cone represents a reverse simulation of the possible trajectories of the robot. The case where X is an edge or polygon is also solved by an equivalent straight-line reachability computation.

Similarly, weak backsliding points from an edge $\bar{e}$ can be computed by pretending that $\bar{e}$ is a start edge for a forward sliding simulation.

5.4.4 Generalized Spring Weak Backprojection

Under the generalized spring trajectory model, the commanded motion is given by a position p. Straight-line weak backprojection points from a backprojection base X under p can be computed by convex visibility analysis using a polygonal lens (Section A.4). The focal point for the visibility computation is set to the commanded position p. The reachable volume for the visibility computation contains all straight-line trajectories that terminate at p and come within ϵ_t of X. These are

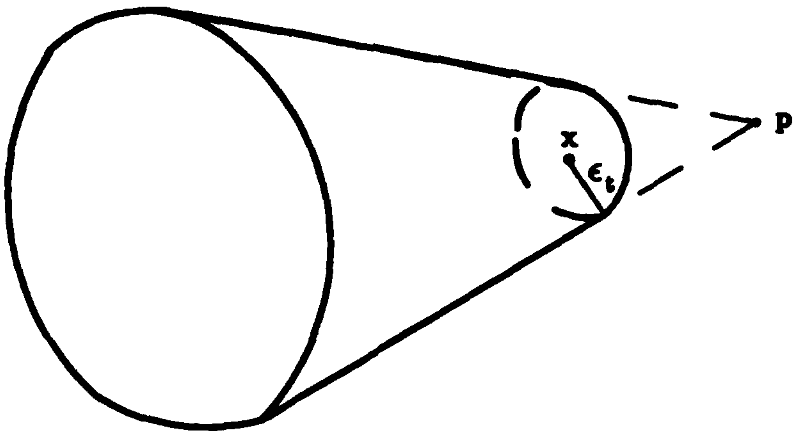


Figure 5.15: The commanded position is **p**. The volume shown contains all trajectories that may strike the point **x** under sensing and control uncertainty.

the trajectories that may strike X under sensing and control uncertainty. Figure 5.15 shows the reachable volume when X is a point. Figure 5.16 shows the reachable volume when X is an edge. Figure 5.17 shows the reachable volume when X is a polygon.

Now consider the computation of weak backsliding points from an edge $\bar{e}$ on a face F under the commanded position $\mathbf{p}$. Recall that sliding trajectories from a contact configuration $\mathbf{s} \in F$ under the generalized spring trajectory model are contained in the projection of a cylinder of radius ϵ_t onto F (Figure 3.14).

Let $\mathbf{p}'$ be the projection of $\mathbf{p}$ onto the plane of F. The weak backsliding region contains all straight-line trajectories on F which aim at $\mathbf{p}'$ and come within ϵ_t of $\bar{e}$, minus any straight-line trajectories that are guaranteed to stick before reaching $\bar{e}$. Figure 5.18 shows the weak backsliding area for one example.

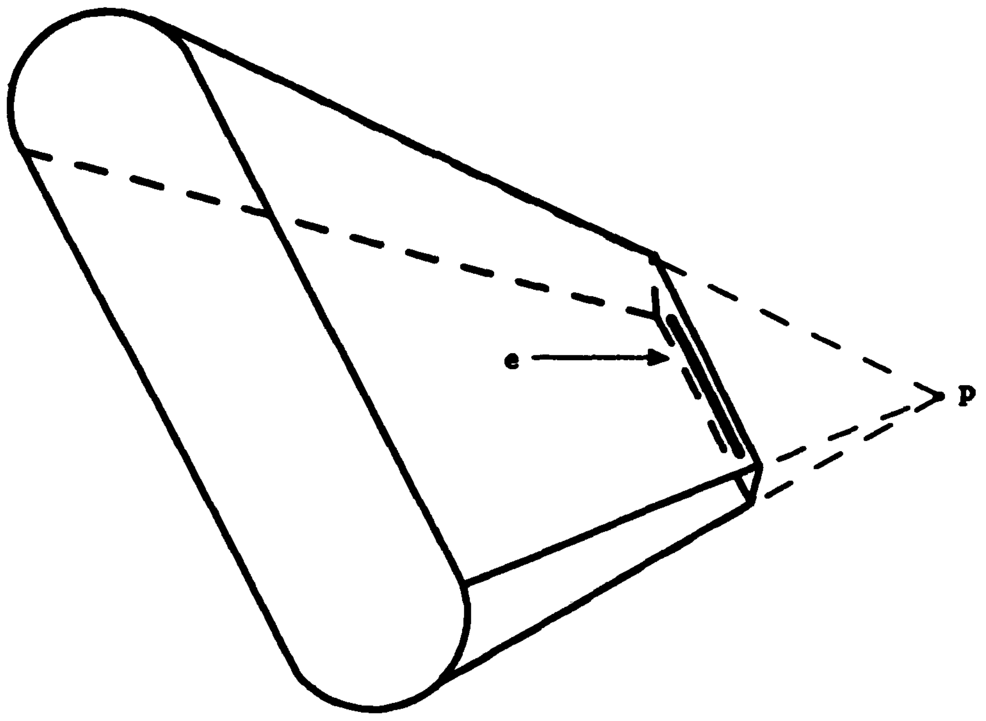


Figure 5.16: The commanded position is **p**. The volume shown contains all trajectories that may strike the edge $\bar{e}$ under sensing and control uncertainty.

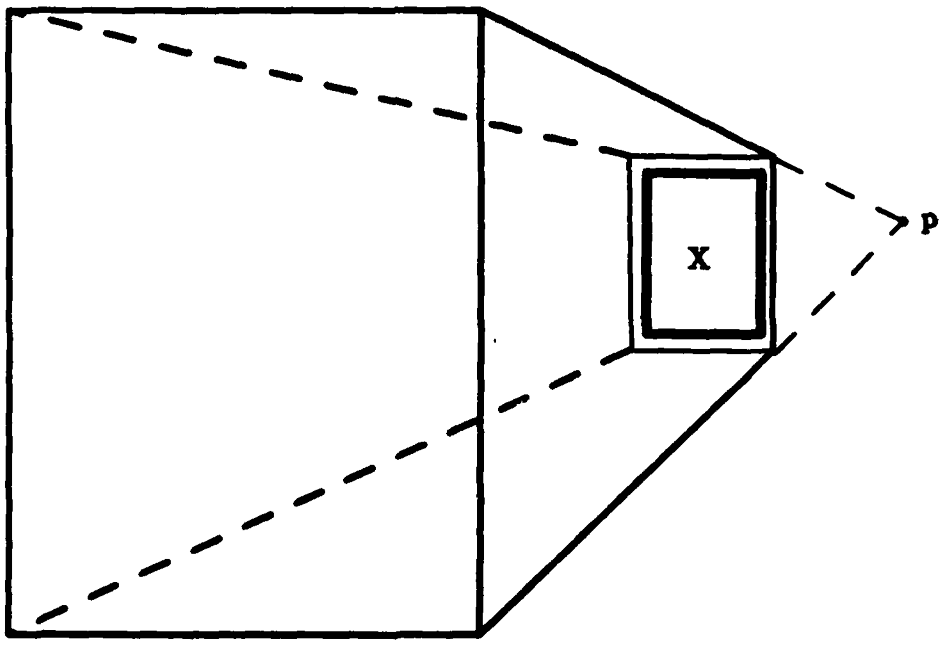


Figure 5.17: The commanded position is **p**. The volume shown contains all trajectories that may strike the polygon X under sensing and control uncertainty.

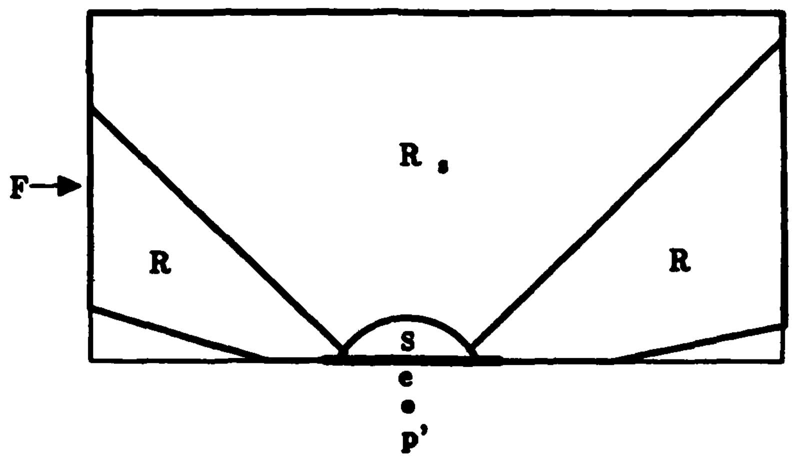


Figure 5.18: A top view of a face F. The projection of the commanded position is p$'$. $\bar{e}$ is an edge segment of F. S is the strong sticking region of F under **p**. R_s is the strong backsliding region of S under **p**. R is the weak backsliding region of $\bar{e}$ under **p**.

Chapter 6

Planning

This chapter presents an algorithm for autonomously planning a compliant motion strategy. The input to the planner is the task geometry, a set of start polygons, and a set of goal polygons. The output of the planner is a compliant motion strategy (Figure 1.6). The input and output are the same as that of the teaching system (Chapter 5). The difference between the two is that the planner does not require any suggested motions from a teacher.

The problem of planning compliant motions with uncertainty is an instance of the general problem of planning motions with uncertainty. Recently, Canny and Reif [1986] showed that the general problem is exponential time hard. The general problem may actually be unsolvable [Erdmann 1984]. In spite of these theoretical constraints, our goal was to implement a working compliant motion planner. The result is an implemented planner which does not always succeed when a solution exists. Experimentation has indicated that the planner can solve a useful class of problems nevertheless.

This chapter is organized as follows:

- Section 6.1 discusses special assumptions that will be made by the planner.
- Section 6.2 presents an overview of the planner.
- Section 6.3 describes some experiments that were conducted with an implemented planner.
- Section 6.4 describes the planning space that we have chosen to represent the state of the robot. Embedded in the planning space is a *state graph*, which makes explicit the possible state transitions of the robot for a particular set of inputs.
- Section 6.5 describes a method for obtaining a reliable motion strategy, given a state graph.
- Section 6.6 describes a method for constructing a state graph, given the planner input.

- Section 6.7 describes a method for simulating the possible motions of the robot. This simulation is used to construct the arcs of a state graph.
- Section 6.8 discusses the computational complexity of the planner.
- Section 6.9 summarizes the chapter.

6.1 Assumptions

The framework of the planner is independent of the trajectory model used. The only part of the planner that relies on the trajectory model is a set of functions which simulate motions. These functions are described in Section 6.7. For simplicity, we will assume the generalized spring trajectory model. To convert to the generalized damper model, one would merely have to replace these functions by equivalent functions which simulate generalized damper motions.

Motions can be terminated in three ways. For a type 1 (sticking) motion, the robot approaches a commanded position, and stops by sticking along the way. For a type 2 (position sensing) motion, the robot approaches a commanded position, and stops when its force sensors detect contact, and its position sensors are contained in a given set of termination positions. For a type 3 (position and force sensing) motion, the robot approaches a commanded position, and stops when its force sensors are contained in a given set of termination forces, and its position sensors are contained in a given set of termination positions. For type 2 and 3 motions, it should be noted that the robot might also stick somewhere along the way, depending on the task geometry.

Since sticking is possible in a type 2 or 3 motion, these motion types can accomplish any task that a type 1 motion can. The reason for studying type 1 motions is that (a) they are simpler, and (b) for compliant motions, one can accomplish quite a bit without sensing termination. Since compliant motions involve contact, sticking is a natural way to terminate a motion. If we were to allow free space goals, then sensing termination would become much more critical.

Nevertheless, there are tasks that a type 2 motion can accomplish but a type 1 motion cannot. Suppose that the goal region is a frictionless face. Sticking on the face requires that the robot issue a velocity which is precisely normal to it. If there is any velocity uncertainty at all, then the robot cannot do this reliably. It must use sensing to recognize that it has reached the goal, and stop on its own accord.

A type 3 motion can accomplish any task that a type 2 motion can, because type 3 motions are a generalization of type 2 motions. In a type 2 motion, the force sensors can only distinguish between contact and free space. In a type 3 motion, the force sensors can also detect the angle of the contact force, to within θ_f. This allows certain surfaces to be distinguished that cannot be by a type 2 motion. For an example, see Figure 6.1.

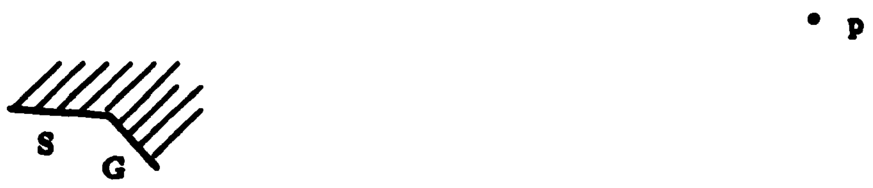


Figure 6.1: An example where type 3 termination is needed. The robot starts in the edge S. The goal region is the edge G. All points in G are within ϵ_s of S, so position sensing cannot disambiguate G from S. The reaction forces of S and G are disambiguable. The strategy is to issue a commanded position which causes sliding on both S and G. Termination occurs when a reaction force of G is recognized.

6.2 Overview of the Planner

This section presents an overview of the planner. The rest of the chapter supplies the details.

An *atom* is a set of contiguous configurations which share a common set of possible reaction forces. Any face, edge, or vertex from the configuration space environment is an atom. All of free space is an atom. The set of all atoms is a partition of configuration space into complete, nonoverlapping subsets. The planner characterizes the state of the robot as a collection of atoms where the robot might be located, under bounded sensing and control uncertainty.

Figure 6.2 shows a simple triangular environment, consisting of three edges and three vertices. The robot can move about the interior of the triangle. There are seven atoms in this environment: three vertices, three edges, and free space. The state of the robot is characterized as a subset of the seven atoms. For example, the state $\{1,2\}$ implies that the robot might be on vertex 1 or edge 2, but that it cannot determine which one it is on due to sensing uncertainty. The state $\{1,2,3\}$ is impossible with reasonable position sensing, for if the robot were in this state, then it could easily distinguish state 1 from state 3, and fix its state more precisely. Similarly, the state $\{2,3,4\}$ is impossible under reasonable force sensing, for if the robot were in this state, then it could easily distinguish state 2 from state 4, and fix its state more precisely. Thus, the only states that we need to consider are states which contain atoms that are confusable with each other under position and force sensing uncertainty.

The planner operates by repeatedly choosing a state, and constructing arcs which connect the state to other states. Arcs represent reliable motions between states. Arc construction proceeds from state to state until a successful compliant motion strategy has been constructed from the start state to a goal state.

To construct arcs from a state, the planner computes all possible commanded motions from the state, grouping them into sets called *weak arcs*, which each have a common target atom. For example, consider the singleton state $X = \{1\}$ from Figure 6.2. We can construct seven weak arcs from X. Each weak arc represents the

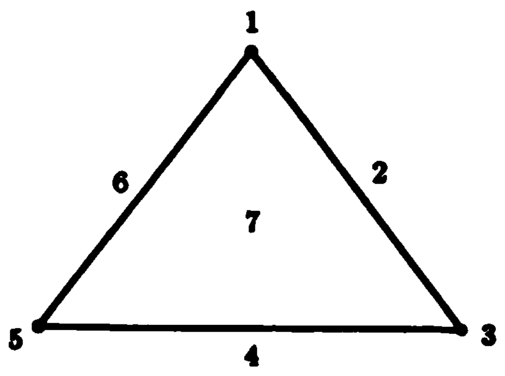


Figure 6.2: A simple planar environment consisting of seven atoms. Atoms 1, 3, and 5 are vertices. Atoms 2, 4, and 6 are edges. Atom 7 represents free space.

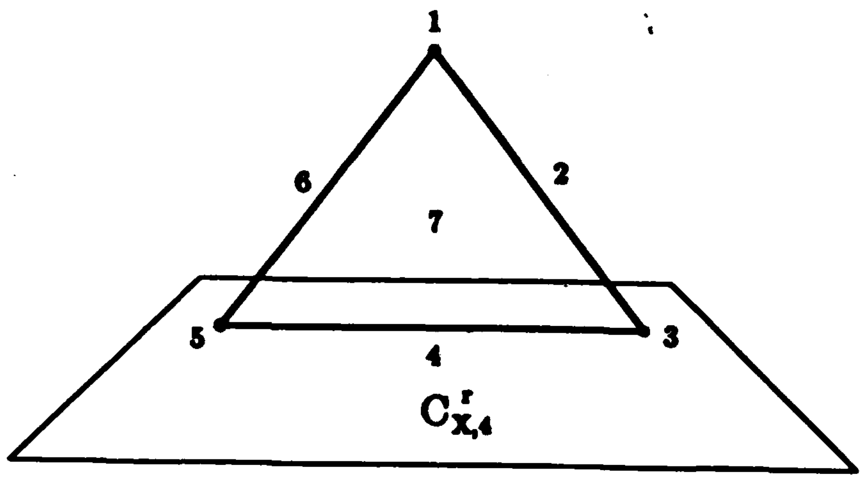


Figure 6.3: The set $C^r_{X,4}$ of commanded positions which might reach atom 4 from the state $X = \{1\}$.

possible commanded positions that might cause the robot to reach and terminate in one of the seven atoms from X. Termination can be by sticking, position sensing, and force sensing. Consider the construction of a weak arc from X to atom 4. Figure 6.3 shows the set $C^r_{X,4}$ of commanded positions which might cause the robot to reach atom 4 from X. Figure 6.4 shows the set C^s_4 of commanded positions which might cause the robot to stick in atom 4 if it reaches it. If we intersect $C^r_{X,4}$ with C^s_4, we obtain the set $C_{X,4}$ of commanded positions which might cause the robot to reach and stick in atom 4 from state X. $C_{X,4}$ generates a weak arc from X to atom 4 under sticking termination.

Weak arcs are not reliable. A commanded position in $C_{X,4}$ may also be in $C_{X,3}$. This implies that the commanded positions in $C_{X,4}$ are not guaranteed to reach and terminate in atom 4. We need a *strong arc*, one which is guaranteed to reach and terminate in its target. Weak arcs can be converted to strong arcs by a

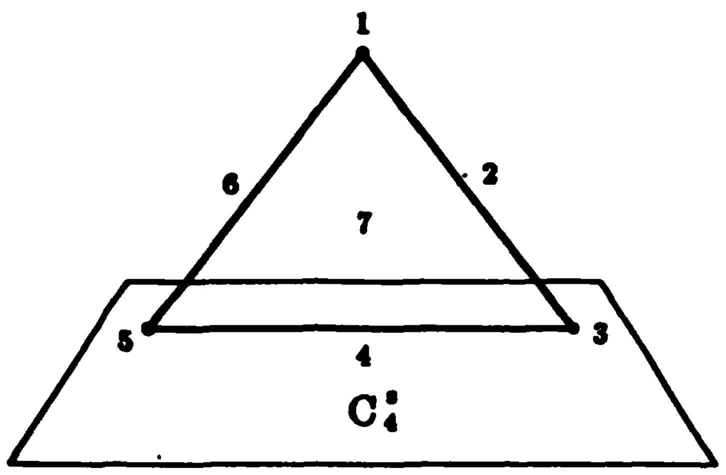


Figure 6.4: The set C_4^s of commanded positions which might cause the robot to stick in atom 4 if it reaches it.

Target Atoms	*Commanded Positions*
1,2	$C_{X,1} \cap C_{X,2}$
1	$C_{X,1} - C_{X,2}$
2	$C_{X,2} - C_{X,1}$

Table 6.1: Composition of the weak arcs $C_{X,1}$ and $C_{X,2}$ under sticking termination.

process called *arc composition*. Assume initially that there are only two atoms in an environment, atom 1 and atom 2. We are to construct the arcs of a state X under sticking termination. First, two weak arcs are constructed, $C_{X,1}$ and $C_{X,2}$. The composition of these two weak arcs is shown in Table 6.1. The first composed arc contains the intersection of the commanded positions of the weak arcs. Its possible target states are $\{1\}$, $\{2\}$, and $\{1,2\}$, assuming that atoms 1 and 2 are confusable. The other two arcs contain the differences of the commanded positions of the weak arcs. The target states of these arcs are $\{1\}$ and $\{2\}$, respectively. Assuming that there are only two atoms in the environment, these three arcs are reliable.

Of course, in real environments, there are more than just two atoms. Any number of weak arcs can be converted to strong arcs by repeated pairwise composition, as long as the initial set of weak arcs contains all possible motions from the source state. Confusable collections of the target atoms of a strong arc form target states that the robot might reach if it follows the arc. On execution, the robot decides which target state it has actually reached using position and force sensing.

Weak and strong arcs can be extended to represent compliant motions which use sensing termination. Assume initially that there are only two atoms in the environment, and that we are to construct the arcs of a state X under position

Target Atoms	*Commanded Positions*	*Sensed Positions*
1,2	$C_{X,1} \cap C_{X,2}$	$S_1 \cap S_2$
1	$C_{X,1} - C_{X,2}$	S_1
1	$C_{X,1}$	$S_1 - S_2$
2	$C_{X,2} - C_{X,1}$	S_2
2	$C_{X,2}$	$S_2 - S_1$

Table 6.2: Composition of the weak arcs $(C_{X,1}, S_1)$ and $(C_{X,2}, S_2)$ under position sensing termination.

sensing termination. First, two weak arcs are constructed. The first weak arc has commanded positions $C_{X,1}$, and terminates on the sensed positions S_1. $C_{X,1}$ is the set of commanded positions which might reach atom 1 from X. S_1 is the set of possible sensed positions of atom 1. The second weak arc has commanded positions $C_{X,2}$, and terminates on the sensed positions S_2. The composition of these two weak arcs is shown in Table 6.2. The first composed arc contains the intersection of the commanded positions and sensed positions of the weak arcs. Its possible target states are $\{1\}$, $\{2\}$, and $\{1,2\}$, assuming that atoms 1 and 2 are confusable. The target state of the second composed arc is $\{1\}$. The composed arc contains the commanded positions of the first weak arc minus the commanded positions of the second weak arc. This ensures that atom 2 is not reachable by the composed arc. The target state of the third composed arc is also $\{1\}$. The composed arc contains the sensed positions of the first weak arc minus the sensed positions of the second weak arc. This ensures that if atom 2 is reached by the arc, the robot does not terminate there by position sensing. The fourth and fifth composed arcs are the analogs of the second and third composed arcs, for the case where the target state is $\{2\}$. For any of these arcs, there is a possibility that the robot may terminate in an intermediate atom due to sticking. This case is handled later in the chapter. The case of force sensing termination is also discussed later in the chapter.

6.3 Examples

In this section, we present some examples that have been run on an implemented planner. The planner is implemented in ZetaLisp, on a Symbolics 3600 computer.

6.3.1 Cube Environment With Sticking Termination

Figure 6.5 shows an environment consisting of a cube. The robot can slide around on the inside faces of the cube, or travel through the free space inside the cube.

The width of the cube is 2 inches. The coefficient of friction of the cube faces is .25. The sensing and control error bounds are as follows:

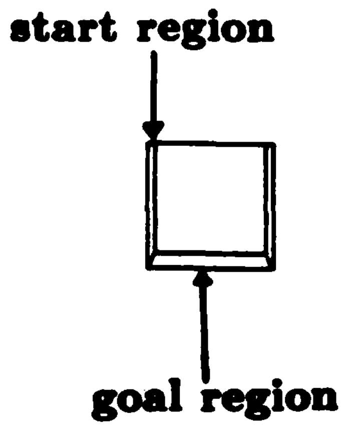


Figure 6.5: An environment consisting of a cube. The robot is inside of the cube. The start region is a top edge of the cube. The goal region is the interior of the cube bottom.

- $\epsilon_s = .2$ inches
- $\theta_f = .3$ radians
- $\epsilon_p = .2$ inches
- $\theta_v = .25$ radians

Under these error bounds, the maximum free space trajectory error ϵ_t given by Equation 3.21 is equal to .218 inches.

The start region is defined as a top edge of the cube, as shown in Figure 6.5. The goal region is the interior of the cube bottom. Sticking termination is to be used. The planner returns a compliant motion strategy consisting of a single commanded motion, as shown in Figure 6.6.

6.3.2 Cube Environment With Position Sensing Termination

We will again use the cube environment. This time, the start region is defined as a bottom edge of the cube, as shown in Figure 6.7. The goal region is again the interior of the cube bottom. Position sensing termination is to be used. The planner returns a compliant motion strategy consisting of a single commanded motion, as shown in Figures 6.8 and 6.9.

6.3.3 Hole Environment With Sticking Termination

For this example, we will use the hole environment described in Section 5.2. We will assume the same start and goal regions that were used in that section. The

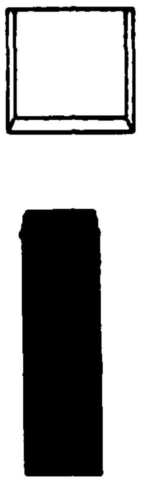


Figure 6.6: The black polyhedron contains commanded positions that solve the cube problem of Figure 6.5. Termination is by sticking on the cube bottom.

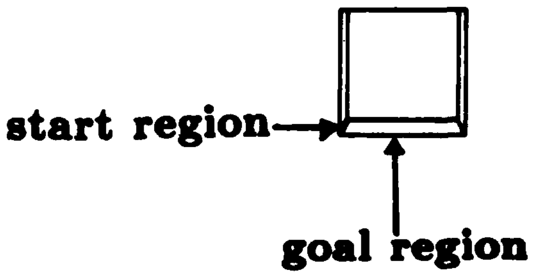


Figure 6.7: The cube environment again. This time, the start region is a bottom edge of the cube, and the goal region is the interior of the cube bottom.

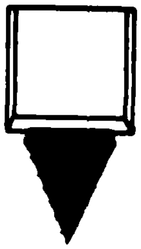


Figure 6.8: The black polyhedron contains commanded positions that solve the cube problem of Figure 6.7. Termination is by sensing the interior of the cube bottom, as shown in Figure 6.9.

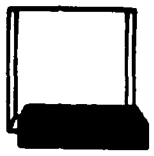


Figure 6.9: The black polyhedron contains sensed positions that terminate the motion of Figure 6.8 in the interior of the cube bottom.

coefficient of friction of the hole surfaces is now .48. The sensing and control uncertainty bounds are the same as in Section 5.2, except for θ_f, which is now .05 radians. Sticking termination is to be used. The planner returns a compliant motion strategy consisting of a sequence of two commanded motions, shown in Figures 6.10 and 6.11. Note the similarity between this strategy and the strategy that we developed in Section 5.2.

6.3.4 Peg-in-Hole Insertion on a Real Robot

The strategy produced by the planner in Section 6.3.3 was tested on the insertion of a rectangular steel peg into a rectangular steel hole, using an IBM 7565 robot. The 7565 is a Cartesian robot with a parallel jaw gripper and a three degree-of-freedom wrist. The wrist maintained a fixed orientation during the insertion. The peg was grasped from a corner of the hole, to ensure that it was rotationally aligned. To eliminate misalignment during the insertion, the following steps were taken:

- The gripper fingers were aligned with the plane of motion, to prevent a reaction force in the direction of motion from rotating the peg within the gripper fingers.

- The peg and gripper fingers were lined with sandpaper, to prevent incidental slippage.

The 7565 is programmed using the AML language [Taylor et al 1982]. The AML program for the experiment is given in Appendix C.

There were a few minor differences between the parameters used in Section 6.3.3 and those of the physical experiment. The parameters of the physical experiment were as follows:

- hole clearance = .025 inches

- hole depth = .05 inches

- hole coefficient of friction = .48

- ϵ_s = .005 inches

- θ_f = .05 radians

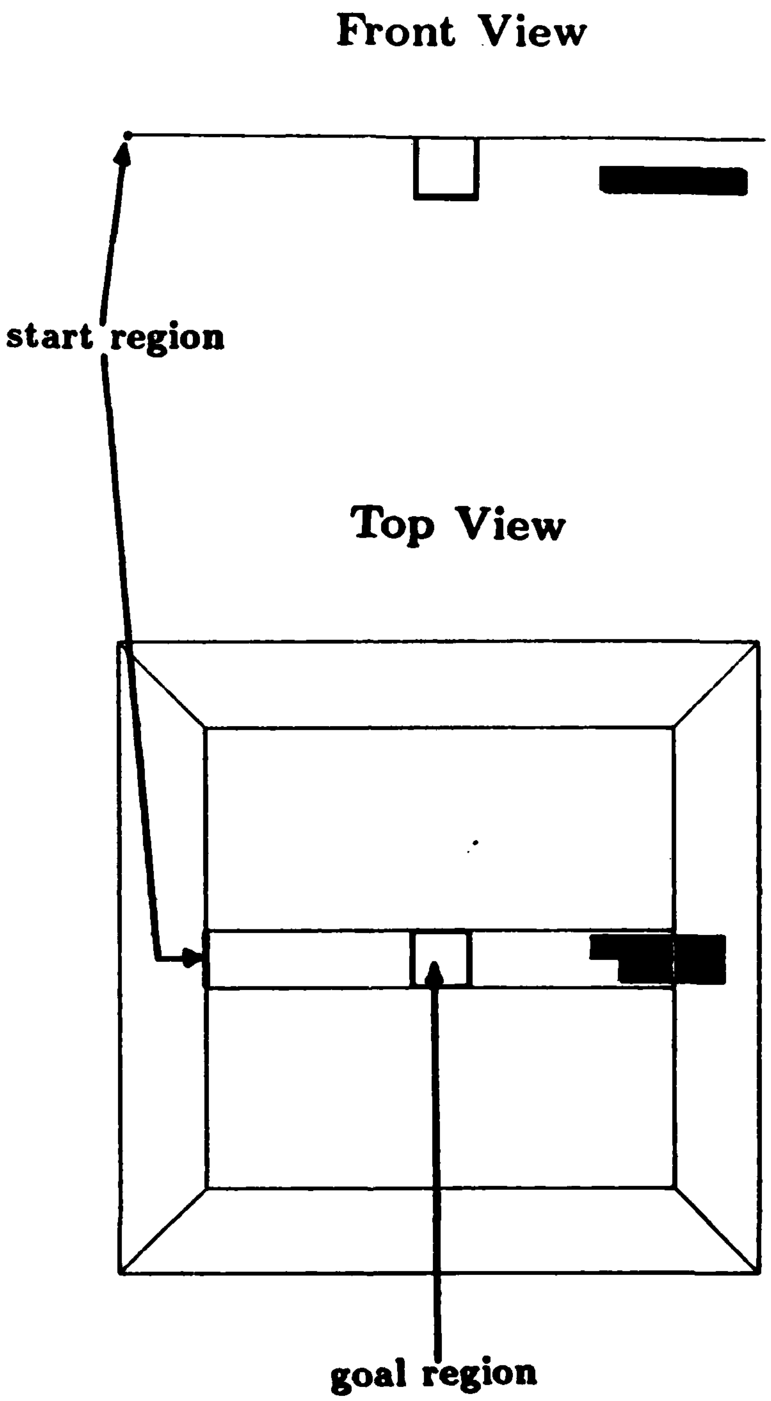


Figure 6.10: Front and top views of the hole environment. The black polyhedra contain commanded positions that comprise the first motion of the strategy that solves the hole problem of Figure 5.2. The robot terminates by sticking on the side face of the hole.

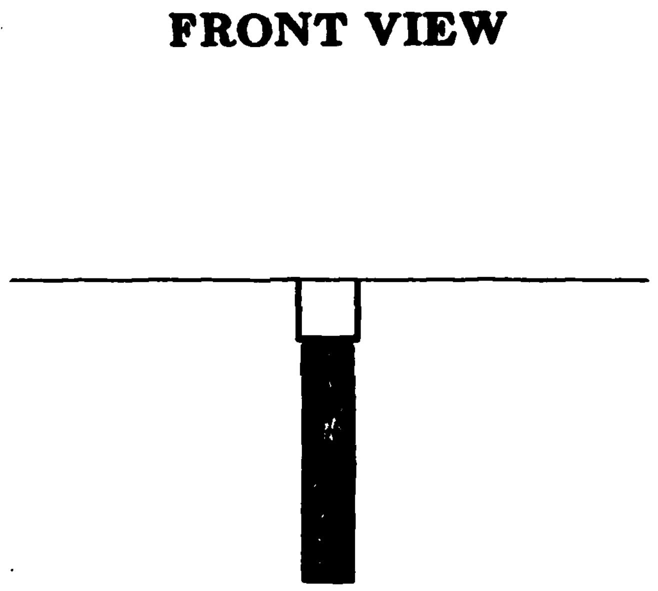


Figure 6.11: A front view of the hole environment. The black polyhedron contains commanded positions that comprise the second and final motion of the strategy that solves the hole problem of Figure 5.2. The robot terminates by sticking on the bottom of the hole.

- $\epsilon_p = .005$ inches
- $\theta_v = .25$ radians

The coefficient of friction between the steel peg and hole was measured experimentally. If we scale the geometry of the experiment by 40, then we obtain the following parameters:

- hole clearance = 1 inch
- hole depth = 2 inches
- coefficient of friction = .48
- $\epsilon_s = .2$ inches
- $\theta_f = .05$ radians
- $\epsilon_p = .2$ inches
- $\theta_v = .25$ radians

These parameters are identical to those used in Section 6.3.3.

Since AML does not supply compliant motion commands, I implemented a slow force feedback loop in AML, using the strain gauges in the robot fingers to measure reaction forces. Given a final commanded position, a trajectory is constructed which consists of a series of closely spaced via points. For each via point $\mathbf{x}_c$, the feedback loop commands the actual position

$$\mathbf{x} = \mathbf{x}_c + \mathbf{f}_r/k, \tag{6.1}$$

where $\mathbf{f}_r$ is the sensed reaction force vector, and k is the stiffness constant. When a transition from contact to free space is detected, the trajectory is replanned, to prevent a large motion from being commanded. A trajectory is terminated when either the final commanded position is reached, or two consecutive forces are sensed which oppose motion with large magnitudes. The latter is assumed to indicate sticking.

The critical motion in the strategy is the first one, in which the peg clears the top of the hole. After the robot makes it into the hole, it is a simple matter to drop to the bottom of the hole, since the peg is rotationally aligned. The insertion was tested using extreme points from the rectangular polyhedra which represent the first motion. All of the test points were successful. The insertion was also tested using points which were exterior to the motion polyhedra. Many of these points were successful; some were not. The successful points are indicative of the conservative nature of the planner.

This particular insertion could have been performed by position control. However, the purpose of the exercise was to test initial feasibility of the compliant motion strategies that are produced by the planner. Additional work is required to develop robust implementations of compliant motion control. Once robust implementations are available, we can investigate the ultimate goal: insertions that cannot be performed by position control, at speeds that rival human assembly speed.

6.3.5 Double Hole Environment With Sticking Termination

Section 1.1 discussed an example involving a double hole environment. The holes of the example had the same clearances as the single hole above. The sensing and control error bounds were also the same as above. Sticking termination was used to solve the example, resulting in a decision-free strategy of two commanded motions. The example showed that the planner can be used on environments in which there are a fairly large number of faces.

6.4 Choosing a Planning Space

The first step in the design of a planner is to choose a *planning space* in which to characterise the state of the robot. A point in this space would represent a state of the robot. The start region and the goal region would be states in this space. When possible, states would be connected by arcs, representing reliable motions between states. The arcs would define a *state graph*, embedded in the planning space. The state graph could then be searched for a path from the start state to the goal state.

One possible choice for the planning space is configuration space. A state in this planning space would be defined as a configuration of the robot. The problem with this choice is that a state of the robot cannot be limited to a single configuration, due to sensing and control uncertainty.

A better choice for a state is a *set* of configurations. This is the type of state that was used by the framework of Losano-Péres, Mason, and Taylor [1984]. Unfortunately, since a configuration space is continuous, it contains an uncountable number of points. The set of states is thus uncountable as well. If we are to design a working planner, we will have to come up with a simpler set of states.

One way to make the number of states countable is to form a uniform cell decomposition of configuration space, and define a state as a union of cells from this decomposition. The cell size could be equal to the commandable increment of the robot. A robot's *commandable increment* is the distance between commandable positions. This is usually determined by the interface between its computer controller and its analog hardware.

The number of cells in a uniform decomposition of configuration space is countably infinite, since configuration space is infinite. We can make the number of cells finite by bounding the environment. In mechanical assembly tasks, this is a reasonable thing to do, because an assembly robot normally has a limited range of motion. The number of states is now finite. Suppose that the number of cells is k_c. Then the number of states is equal to 2^{k_c}. This is a huge number.

To further reduce the number of states, we could decrease the resolution of the uniform cell decomposition. We could also abandon the idea of having cells of uniform size. Instead, we could partition configuration space into nonoverlapping subsets called *atoms*, whose union is the configuration space. We would treat each atom as we would a cell. If the robot arrives at a configuration in an atom A, then the planner assumes that it could be anywhere within A.

A useful definition for an atom is a set of contiguous points which share a common set of possible reaction forces. This is a good choice because it makes it easy to predict the possible behavior of the robot regardless of where it is within an atom. Any face, edge, or vertex qualifies as an atom under this definition. Free space also qualifies as an atom. A face is considered to be an open set, bounded by edges and vertices. Similarly, an edge is considered to be an open set, bounded by two vertices. These definitions make the decomposition nonoverlapping. It should be noted that this is not the only possible choice for the atom decomposition. More will be said about this later in this section.

The number of states is now equal to 2^{k_a}, where k_a is the number of atoms. This is still unmanageable for most environments. A further reduction can be obtained by pruning unnecessary states. Note that two atoms can often be disambiguated from each other by position and force sensing. This observation is formalized in the following definition:

Definition 6.1 *Suppose that A_1 and A_2 are atoms. A_1 and A_2 are* **disambiguable** *if either of the following conditions holds:*

1. *The closest points of A_1 and A_2 are greater than $2\epsilon_s$ apart.*
2. *The closest pair of possible reaction force vectors of A_1 and A_2 differ by an angle which is greater than $2\theta_f$.*

In the first case, if the robot is in A_1, then the maximum distance between the sensed position and A_1 is ϵ_s. The maximum distance between an interpretation of the sensed position and A_1 is then $2\epsilon_s$. Since every point of A_2 is greater than $2\epsilon_s$ from A_1, there is no way that the sensed position can be interpreted as belonging to A_2.

In the second case, if the robot is in A_1, then the maximum angle between the sensed reaction force and the possible reaction forces of A_1 is θ_f. The maximum angle between an interpretation of the sensed reaction force and the possible reaction forces of A_1 is then $2\theta_f$. Since every reaction force of A_2 is greater than $2\theta_f$ from those of A_1, there is no way that the sensed reaction force can be interpreted as belonging to A_2.

If the robot reaches the state $\{A_1, A_2\}$, and A_1 and A_2 are disambiguable, then it can make a transition to either $\{A_1\}$ or $\{A_2\}$ without issuing a motion. More generally, any state X of n atoms which contains a disambiguable pair of atoms A_1 and A_2 can be subdivided into two states of $n - 1$ atoms each, with A_1 going into one of the substates, and A_2 going into the other. If the robot reaches X, then it can make a motionless transition to one of the substates. It can continue making motionless transitions until it reaches a substate with no disambiguable pair of atoms. A state with no disambiguable pair of atoms is said to be *primitive*.

Currently, a state arc represents a reliable motion from one state to another. It is labeled by a commanded motion. The target state contains all possible next atoms. We would like the robot to proceed directly to the primitive substate which contains all atoms that are consistent with the sensor readings at execution time. To

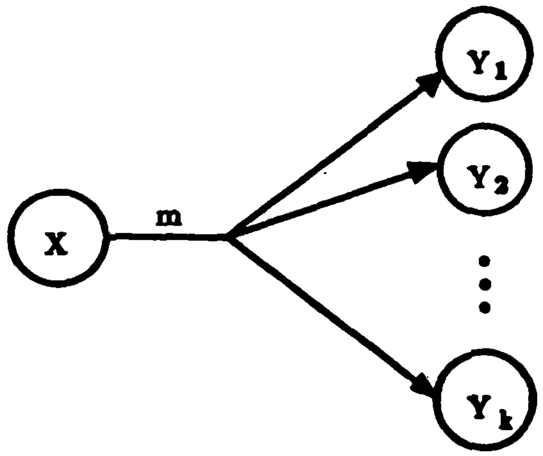


Figure 6.12: An arc of a state graph. The robot begins the motion m in state X. It terminates in one of the states $\{Y_i\}$.

do this, we need to replace the standard single-headed arc with a multi-headed arc, as shown in Figure 6.12. Each head of the new arc points to a primitive state. On the completion of the commanded motion, the robot computes the set of atoms that are consistent with the sensor readings, and proceeds directly to the corresponding primitive state.

In practice, we have found it necessary to reduce the number of states further, by limiting the number of atoms per state. We found it adequate for simple examples to concentrate on states with three or less atoms.

Table 6.3 shows the reduction in the number of states when we limit them to primitive states, and when we limit the number of atoms per state. We generated the states of three environments, under fairly large sensing uncertainty.

With the splitting of disambiguable states, the start and goal regions might now be sets of states. Our job now is to find a motion strategy which reliably terminates in a goal state from each start state. Since states are disambiguable, the start states can be treated as separate planning problems. The robot need only determine at execution time which start state it is in, and choose the motion strategy that has been planned for that start state. For simplicity, we will assume from now on that there is a single start state, and multiple goal states.

6.4.1 Computing Primitive States

Suppose that we are given a set Σ of atoms. We are to construct the set of primitive states that consist of atoms from Σ.

We certainly want to avoid a generate-and-test solution. Suppose that there are n atoms in Σ. A generate-and-test algorithm would enumerate all 2^n subsets of the atoms, and test whether each one is primitive.

We can avoid the exhaustive solution by careful pruning. The basic principle

environment	cube	hole	double hole
clearance	1 in	1 in	1 in
friction coefficient	.25	.25	.25
ϵ_s	.2 in	.2 in	.2 in
θ_f	.3 rad	.3 rad	.3 rad
faces	6	14	26
atoms	26	66	118
all states	7E7	7E19	3E35
primitive states	258	1714	3082
states $\leq$ 3 atoms	226	934	1694
states $\leq$ 2 atoms	122	386	708
states 1 atom	26	66	118

Table 6.3: The number of atoms, states, primitive states, and states with a limited number of atoms, for three environments under the indicated clearances and sensing uncertainties. Note: the clearance of the double hole applies to both holes.

is this: If a nonprimitive set of atoms is found, then we should not generate any further states which contain this set as a subset. This principle can be put to action by the following algorithm:

1. Create a state S_1 consisting of the first atom in Σ.
2. Call the algorithm recursively to compute the primitive states that can be constructed from the rest of the states in Σ. Set Γ_1 to the new states.
3. Try to form new primitive states by adding S_1 to each state in Γ_1, and testing whether the new state is primitive. Set Γ_2 to the new states.
4. Return S_1, Γ_1, and Γ_2.

This algorithm does not allow the first atom to be combined with any nonprimitive sets of atoms. Thus, it obeys the pruning principle. Note that every time we test whether a state is primitive, we are adding one atom to an already primitive state. Thus, to test whether a new state is primitive, we simply make sure that the new atom is confusable with each of the atoms of the primitive state.

Recall that an atom is either a vertex, an edge, or a polygon. To test whether two atoms are confusable, we test whether their closest points are within $2\epsilon_s$. Then, we test whether their sets of possible sensed forces intersect. Section 6.7.7 describes a method for constructing the set of possible sensed forces of an atom. The force test can be avoided if two atoms share a common face.

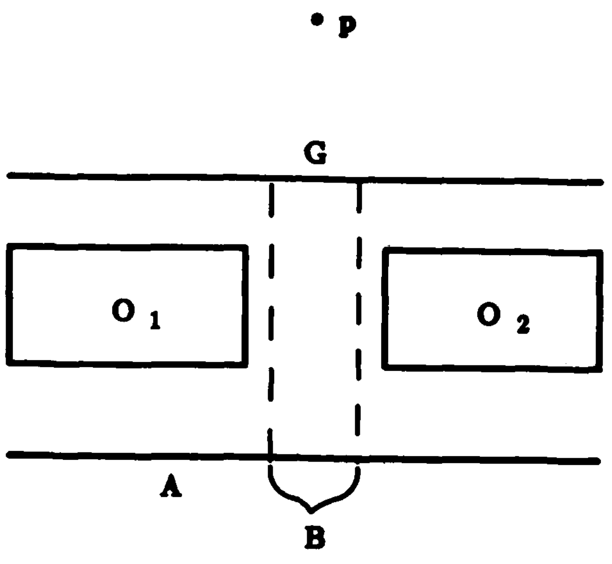


Figure 6.13: An example where the planner is incomplete. The robot is known to be somewhere in the edge segment B. B is contained in the edge atom A. The goal is the edge atom G. The obstacles O_1 and O_2 have infinite friction, so sliding is impossible on them. Although the forward projection of **p** from B does not intersect the obstacles, the planner forward projects from A, and concludes that **p** is unreliable.

6.4.2 A Tradeoff Between Completeness and Efficiency

At the heart of the planning space is the decomposition of configuration space into atoms. The choice of the atom decomposition affects the completeness of the planner, but not its correctness. A motion planner is *correct*, or *reliable*, if the motion strategies that it constructs are guaranteed to work under sensing and control uncertainty. Our planner is correct. A motion planner is *complete* if it is able to find a motion strategy when one exists for a problem. Our planner is complete in the sense that if a motion strategy exists for a particular input, then the planner can find a motion strategy, using a uniform atom decomposition in which atoms are smaller than the commandable increment of the robot. Unfortunately, this scheme would take much longer than anyone would be willing to wait.

Choosing larger atoms, while more efficient, is not complete, since it amplifies the position uncertainty of the robot. Figure 6.13 shows an example where this occurs. Thus, the atom decomposition is a key factor in the success of the planner.

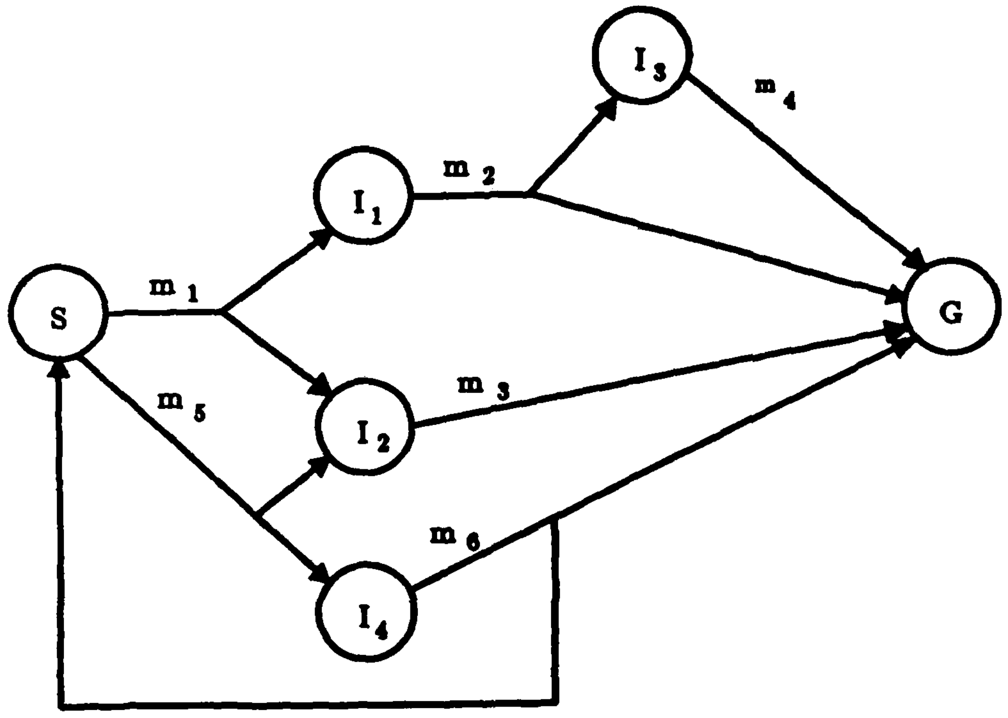


Figure 6.14: An example of a state graph.

6.5 Obtaining a Motion Strategy From a State Graph

A state graph represents the allowable state transitions of the robot. Figure 6.14 shows an example of a state graph. Section 6.6 will discuss how to construct a state graph from the planner input. For now, assume that a state graph has been constructed. How can we obtain a reliable motion strategy from it? A standard search technique, such as depth-first search, won't work, since the arcs in a state graph are multi-headed, representing conditional branches.

6.5.1 Evaluating a State Graph

Consider the question of whether the robot can terminate in a goal state, when starting in a particular state X. We will refer to this as the *success* of X. The following observations can be made: A state is successful if it is a goal state, or one of its arcs is successful. An arc is successful if all of its next states are successful.

These observations can be translated almost directly into recursive procedures for evaluating the success of a state. To test the success of a state graph, we simply evaluate the start state.

One snag is that a state graph may contain cycles. We have to avoid cycling in the graph while testing for success. To do this, we mark a state as a failure when

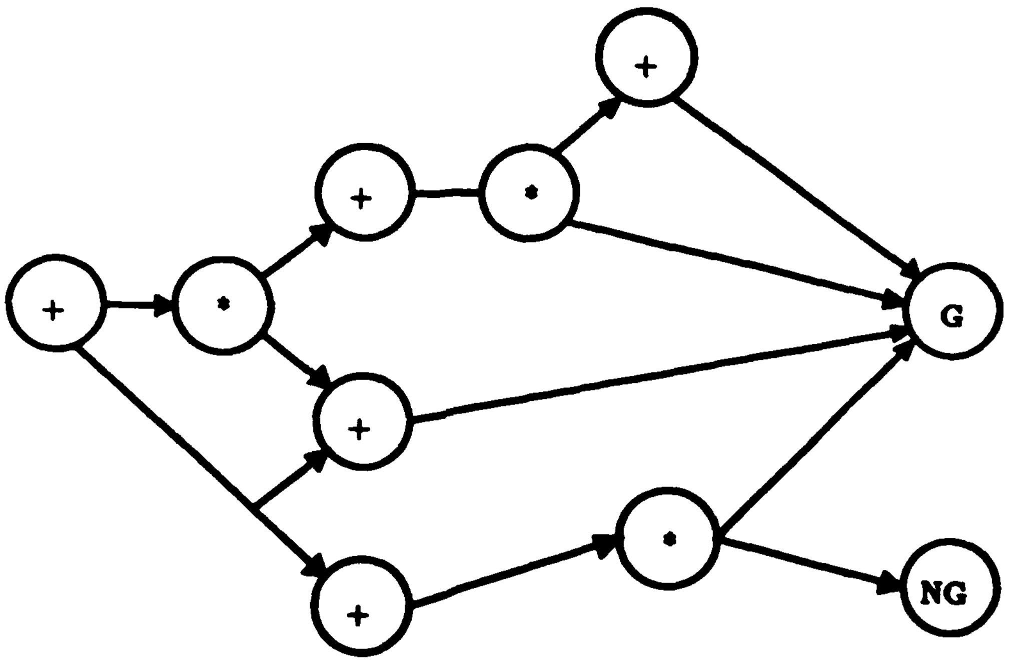


Figure 6.15: An example of an AND/OR tree. AND nodes are labeled with *. OR nodes are labeled with +. Terminal nodes are labeled *G* for goal, or *NG* for nongoal.

it is reached for the first time. Then, if one of its arcs is successful, its evaluation is changed to indicate success. If a state marked as a failure is reached again, then the arc that reaches it fails. Reaching a failed state indicates one of two things. First, the state may have been evaluated as a failure previously. Second, the state may be currently active, meaning that a cycle has been detected. In this case, failure is justified because of our assumption that motion strategies do not contain loops.

6.5.2 A State Graph is Like an AND/OR Tree

An AND/OR tree [Winston 1984] is a tree which has two types of nodes, AND nodes and OR nodes. Figure 6.15 shows an example of an AND/OR tree. Terminal nodes of the tree can be evaluated statically as either *goal* or *nongoal*. An AND node is successful if all of its sons are successful. An OR node is successful if any of its sons are successful. The success of an AND/OR tree is given by the success of its root node.

A state graph is like an AND/OR tree. To see this, consider Figure 6.16, which shows an alternative representation of the state graph arc of Figure 6.12. The source state X is depicted as an OR node. X is shown pointing at an AND node, which points at the target states $\{Y_i\}$. The AND/OR representation makes the evaluation procedure of Section 6.5.1 explicit.

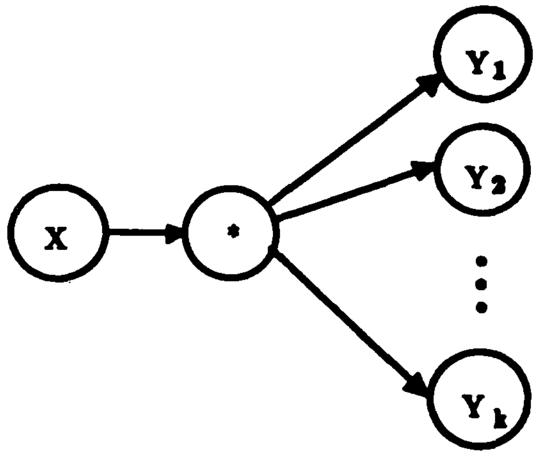


Figure 6.16: An alternative view of the state graph arc of Figure 6.12. The source state X is seen as an OR node. The arc labeled by * is an AND node.

6.5.3 The Representation of a Motion Strategy

It can now be pointed out that the representation of a motion strategy is a state graph which has been successfully evaluated. Failed nodes and arcs can be discarded. Figure 1.6 is the result of removing failed nodes and arcs from the state graph of Figure 6.14.

The resulting state graph is suitable for execution on a robot. The robot executes the commanded motion which labels the arc of the start state. It then decides which next state it has terminated in, using sensors. If the new state is a final state, then the robot stops. Otherwise, it continues the execution/sensing cycle. The robot is guaranteed to terminate in a final state.

6.6 Constructing a State Graph

Now that we know how to obtain a motion strategy from a state graph, we are faced with the task of constructing a state graph from planner input. This involves constructing an arc between a state and a set of states whenever a corresponding motion is possible. We must decide how to construct arcs, and in what order. Section 6.6.1 discusses the choice of a state to expand. Section 6.6.2 describes an algorithm for constructing arcs.

6.6.1 Choosing a State to Expand

An arc is a mapping from a state to a set of states. Because arcs fan out, it is more convenient expand states in the forward direction than the backward direction. That is, given a state, we will construct the arcs that emanate from it, rather than those that terminate at it. We start with a state graph which consists of the start state and the goal states. A best first strategy is used to pick an unexpanded nongoal state in the graph to expand. Currently, the planner picks the unexpanded nongoal state which is closest to a goal state. After constructing the new states and arcs that are generated by the expansion, the resulting state graph is evaluated. If the test returns success, then the planner returns the resulting state graph. Otherwise, the planner continues to expand states. This approach returns the first reliable motion strategy that can be found. To find the best motion strategy (with respect to some cost function), the planner would have to enumerate many strategies, evaluating the cost of each one. This would take significantly longer, although it would not be impossible in principle.

6.6.2 Expanding the Arcs of a State

Expanding a state means constructing its arcs. First, we will discuss how to compute the arcs of a singleton state (a state with only one atom). Then, we will extend the algorithm to states with multiple atoms.

Expanding the Arcs of a Singleton State

Suppose that a state contains a single atom A. We are to compute the reliable motions that can be commanded from A. One approach would be to enumerate all possible motions. For each motion m, we could compute all atoms where m might terminate, using forward projection. (See Chapter 4.) However, this method is infeasible, since the number of motions is uncountable.

Note that the number of atoms is much smaller than the number of motions. Suppose that we enumerate atoms, rather than motions. For each atom B, we construct the set of all motions that might terminate in B when starting from A. An algorithm to do this is described in Section 6.7. This algorithm can be thought of as a simulation of the possible motions that can be commanded from A. The simulation returns a list of *weak arcs*. An arc is comprised of the following fields:

type This field indicates whether the arc represents motions which terminate by sticking (1), position sensing (2), or position and force sensing (3).

target The set of atoms where the robot may terminate.

cpositions The set of possible commanded positions. At execution time, the robot need only choose one commanded position from this set. [1]

spositions The set of sensed positions that the robot will interpret as signals to terminate.

sforces The set of sensed forces that the robot will interpret as signals to terminate.

[1] For the generalized damper trajectory model, this field would contain the set of possible commanded velocities.

In this case, the target of each weak arc is a single atom. Each weak arc consists of motions which might terminate in the target.

We are not done yet, for the weak arcs are not reliable. The reason is that two weak arcs may intersect, i.e., there may be motions that are common to both arcs. For type 1 arcs, intersection means that the cpositions intersect. For type 2 arcs, intersection means that both the cpositions and the spositions intersect. For type 3 arcs, intersection means that the cpositions, spositions, and sforces all intersect. A commanded motion common to two arcs may terminate in the target atom of either arc, so neither arc can be guaranteed to terminate in its target atom. We need to regroup the motions into a new set of nonoverlapping *strong arcs*. The motions within a strong arc share a common set of target atoms. Intuitively, a strong arc is reliable because each motion in it is not contained in any other arc, so the target atoms of the arc are the only atoms that the motions can reach.

We now present some definitions that will allow us to formalize the notion of reliability. The formalization will enable us to develop a provably correct method for computing a reliable set of arcs.

Definition 6.2 *A arc is said to be* **reliable** *with respect to an atom A if its commanded motions may cause termination only in its targets when starting from A.*

A set of arcs is reliable if each arc in the set is reliable.

Definition 6.3 *A set Ψ of arcs is said to be* **complete** *with respect to an atom A if, for each motion m that is possible from A, each possible target of m from A is contained in some arc of Ψ along with m.*

We can now state sufficient conditions for the reliability of an arc:

Theorem 6.1 *A set Ψ of arcs is reliable with respect to an atom A if Ψ is complete with respect to A, and nonoverlapping.*

Proof: Suppose that Ψ is complete with respect to A, and nonoverlapping. Let m be a possible motion from A. By completeness, m is in some arc $\alpha \in \Psi$. Also, all of the possible targets of m are in arcs that m is in. Because Ψ is nonoverlapping, α is the only arc that m is in. This implies that all of the possible targets of m are in α. Therefore, m is a reliable motion, and α is a reliable arc, for each m and α in Ψ. ∎

For type 1, the simulation of Section 6.7 returns a complete set of arcs. Theorem 6.1 tells us that to make this set reliable, we need to make the set nonoverlapping, while preserving its completeness. This process is called *arc composition*.

Note that the converse of Theorem 6.1 is false. Suppose that Ψ is reliable with respect to A. Is Ψ then complete and nonoverlapping? Ψ may not be complete, because it need not contain all possible motions from A. Ψ may be overlapping, as long as the arcs which overlap have identical targets. This last observation is stated in the following theorem:

Arc	*Cpositions*	*Target*
$\alpha_1\alpha_2$	$C_1 \cap C_2$	$T_1 \cup T_2$
α_1^-	$C_1 - C_2$	T_1
α_2^-	$C_2 - C_1$	T_2

Table 6.4: Composition of a pair of type 1 arcs $\alpha_1 = (C_1, T_1)$ and $\alpha_2 = (C_2, T_2)$.

Theorem 6.2 *A set Ψ of arcs is reliable with respect to an atom A if Ψ is complete with respect to A, and nonoverlapping, except between pairs of atoms which have identical targets.*

Proof: Suppose that Ψ is a set of arcs that is complete with respect to a start atom A, and nonoverlapping, except between pairs of arcs which have identical targets. Consider a pair of overlapping arcs α_1 and α_2 of Ψ. Although α_1 and α_2 intersect, composing them would not result in any new target sets. (See below.) Since Ψ is complete with respect to A, and nonoverlapping between arcs with differing targets, there are no other targets for the motions of α_1 and α_2. Thus, α_1 and α_2 must be reliable with respect to A, and furthermore, Ψ must be reliable with respect to A. ∎

Type 1 Arcs

For type 1, the simulation of Section 6.7 returns a complete set of arcs. We now need to make this set nonoverlapping, while maintaining completeness, as per Theorem 6.1.

Suppose that we have a complete pair of type 1 arcs α_1 and α_2 with respect to an atom A. Denote the cpositions of α_1 by C_1, and the targets by T_1. Denote the cpositions of α_2 by C_2, and the targets by T_2. We can compose the pair by computing the three arcs shown in Table 6.4.

The following theorem establishes the reliability of pairwise type 1 arc composition:

Theorem 6.3 *Suppose that α_1 and α_2 are a complete pair of type 1 arcs with respect to an atom A. The arc set shown in Figure 6.4,*

$$\Psi = \alpha_1 * \alpha_2 = \{\alpha_1\alpha_2, \alpha_1^-, \alpha_2^-\}, \qquad (6.2)$$

is reliable with respect to A.

Proof: The commanded motions of a type 1 arc are given by its cpositions. It can be seen from Table 6.4 that the union of the cpositions of Ψ is equal to $C_1 \cup C_2$. Since the set $\{\alpha_1, \alpha_2\}$ is complete with respect to A, Ψ contains all possible cpositions with respect to A.

The following observations make it clear that every cposition in Ψ is accompanied by all of its possible targets:

- The cpositions $C_1 \bigcap C_2$ were in both α_1 and α_2. $T_1 \bigcup T_2$ thus represents all possible targets of $C_1 \bigcap C_2$. Note that $C_1 \bigcap C_2$ and $T_1 \bigcup T_2$ are together in $\alpha_1\alpha_2$.
- The cpositions $C_1 - C_2$ were in α_1 and did not intersect α_2. Thus, T_1 represents all possible targets of $C_1 - C_2$. Note that $C_1 - C_2$ and T_1 are together in α_1^-.
- The cpositions $C_2 - C_1$ were in α_2 and did not intersect α_1. Thus, T_2 represents all possible targets of $C_2 - C_1$. Note that $C_2 - C_1$ and T_2 are together in α_2^-.

Since Ψ contains all possible motions with respect to A, and every cposition in Ψ is accompanied by all of its possible targets, Ψ is complete with respect to A.

The intersection of the cpositions of Ψ is empty, as can be seen from Table 6.4. Thus, Ψ is nonoverlapping.

Since Ψ is complete with respect to A, and nonoverlapping, it is reliable with respect to A, by Theorem 6.1. ▮

How can pairwise composition be extended to an arbitrary set of arcs Λ? To do this, we will maintain two lists of arcs, Ψ_1 and Ψ_2. Ψ_1 is a list of arcs to be inserted into Ψ_2, and is initialized to Λ. Ψ_2 should always be nonoverlapping, and is initially empty. The union of Ψ_1 and Ψ_2 should always be complete. An arc α is removed from Ψ_1 and inserted into Ψ_2 by composing α with each arc β of Ψ_2. Specifically, this involves the following steps for each β:

1. Compose α with β, producing $\alpha\beta$, α^-, and β^-.
2. Replace β in Ψ_2 by $\alpha\beta$ and β^-.
3. $\alpha \leftarrow \alpha^-$.

Once we have composed α with each β in Ψ_2, we insert α into Ψ_2. (Note that α is updated after each composition.) When Ψ_1 becomes empty, Ψ_2 is a complete, nonoverlapping set.

Type 2 Arcs

For type 2 arcs, the simulation of Section 6.7 does not return a complete set of arcs. Each arc returned by the simulation represents the set of motions that might terminate in a given atom by sensing. The simulation does not take sticking into account. The robot might stick in an intermediate atom on the way to its intended target.

The set of type 2 arcs returned by the simulation can be made complete by composing it with a complete set of type 1 arcs. Suppose that α_1 is a type 1 arc, and α_2 a type 2 arc. Denote the cpositions of α_1 by C_1, and the targets by T_1.

Arc	*Cpositions*	*Spositions*	*Target*
$\alpha_1\alpha_2$	$C_1 \bigcap C_2$	S_2	$T_1 \bigcup T_2$
α_2^-	$C_2 - C_1$	S_2	T_2

Table 6.5: Composition of a type 1 arc $\alpha_1 = (C_1, T_1)$ with a type 2 arc $\alpha_2 = (C_2, S_2, T_2)$.

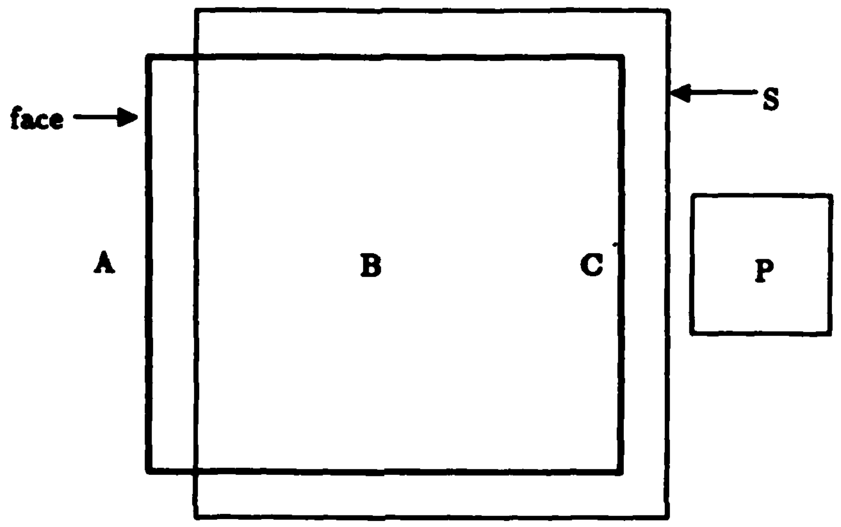


Figure 6.17: This is a top view of a face. A and C are edges of the face, and B is the interior of the face. The robot starts in A. An arc is created with cpositions P, and spositions S. Under P and S, the robot is guaranteed to reach B from A, and stop. C is a sticking region under P, but is not reachable from A. The planner cannot know this using arc composition alone.

Denote the cpositions of α_2 by C_2, the spositions by S_2, and the targets by T_2. The composition of α_1 and α_2 produces the two type 2 arcs shown in Table 6.5.

Composing type 1 arcs with type 2 arcs has no effect on the type 1 arcs. We are merely using the results of the type 1 simulation to add targets to the type 2 arcs. Performing this composition is conservative, since a sticking region which is unreachable may be inserted into a type 2 arc as a target. For example, see Figure 6.17. It is difficult to avoid this without performing a forward projection of the motions in the arc. Leaving a sticking region out of an arc is even worse, for this might make the arc unreliable.

Once we have composed the results of the type 2 simulation with the results of the type 1 simulation, we have a complete set of type 2 arcs. We now need to make this set nonoverlapping, while maintaining completeness, as per Theorem 6.1. Let α_1 and α_2 be a complete pair of type 2 arcs with respect to an atom A. Their

Arc	*Cpositions*	*Spositions*	*Target*
$\alpha_1\alpha_2$	$C_1 \bigcap C_2$	$S_1 \bigcap S_2$	$T_1 \bigcup T_2$
α_1^{c-}	$C_1 - C_2$	S_1	T_1
α_1^{s-}	C_1	$S_1 - S_2$	T_1
α_2^{c-}	$C_2 - C_1$	S_2	T_2
α_2^{s-}	C_1	$S_2 - S_1$	T_2

Table 6.6: Composition of a pair of type 2 arcs $\alpha_1 = (C_1, S_1, T_1)$ and $\alpha_2 = (C_2, S_2, T_2)$.

composition produces five arcs. The first arc is the intersection of α_1 and α_2; the respective cpositions and spositions of α_1 and α_2 are intersected, and the targets are unioned. The second and third arcs result from modifying α_1 to avoid the targets of α_2. The second arc avoids the targets of α_2 by clipping cpositions from α_1 which are in α_2. The third arc avoids the targets of α_2 by clipping spositions which are in α_2. The targets of the second and third arcs are the unmodified targets of α_1. The fourth and fifth arcs are analogs of the second and third arcs, for the case where α_2 is modified to avoid the targets of α_1. The targets of the fourth and fifth arcs are the unmodified targets of α_2. These five arcs are shown in Figure 6.6.

The above intuition can be verified using a more analytical approach. Denote the cpositions of α_1 by C_1, the spositions by S_1, and the targets by T_1. Denote the cpositions of α_2 by C_2, the spositions by S_2, and the targets by T_2. The commanded motions of a type 2 arc are given by pairs of the form (c, s), where c is a cposition, and s is an sposition. c can come from one of three nonoverlapping sets, $C_1 \bigcap C_2$, $C_1 - C_2$, and $C_2 - C_1$. Similarly, s can come from one of three nonoverlapping sets, $S_1 \bigcap S_2$, $S_1 - S_2$, and $S_2 - S_1$. Crossing these two sets produces nine possible pairings of sets. For a pair of type 2 motions to be equal, they must have the same cposition and sposition. These nine set pairings are mutually nonoverlapping, since they were generated by crossing two sets of nonoverlapping sets. Two of the set pairings do not contain motions that are in either α_1 or α_2:

$$(C_1 - C_2, S_2 - S_1)$$
$$(C_2 - C_1, S_1 - S_2)$$

Thus, there are seven feasible pairings of motion sets. These pairings generate the seven arcs shown in Table 6.7. These arcs are complete, because we generated all feasible motion set pairings from α_1 and α_2, and we carried with them all possible targets. They are reliable because they are complete and nonoverlapping.

When two type 2 arcs α_1 and α_2 have identical cpositions and targets, they can be merged into a single type 2 arc whose spositions are the union of the spositions of α_1 and α_2. Similarly, when two type 2 arcs α_1 and α_2 have identical spositions and targets, they can be merged into a single type 2 arc whose cpositions are the

Arc	*Cpositions*	*Spositions*	*Target*
$\alpha_1\alpha_2$	$C_1 \cap C_2$	$S_1 \cap S_2$	$T_1 \cup T_2$
α_1^-	$C_1 - C_2$	$S_1 - S_2$	T_1
α_1^{c-}	$C_1 - C_2$	$S_1 \cap S_2$	T_1
α_1^{s-}	$C_1 \cap C_2$	$S_1 - S_2$	T_1
α_2^-	$C_2 - C_1$	$S_2 - S_1$	T_2
α_2^{c-}	$C_2 - C_1$	$S_1 \cap S_2$	T_2
α_2^{s-}	$C_1 \cap C_2$	$S_2 - S_1$	T_2

Table 6.7: Composition of a pair of type 2 arcs $\alpha_1 = (C_1, S_1, T_1)$ and $\alpha_2 = (C_2, S_2, T_2)$.

Arc	*Cpositions*	*Spositions*	*Sforces*	*Target*
$\alpha_1\alpha_3$	$C_1 \cap C_3$	S_3	F_3	$T_1 \cup T_3$
α_3^-	$C_3 - C_1$	S_3	F_3	T_3

Table 6.8: Composition of a type 1 arc $\alpha_1 = (C_1, T_1)$ with a type 3 arc $\alpha_3 = (C_3, S_3, F_3, T_3)$.

union of the cpositions of α_1 and α_2. Iteratively applying these simplifications to Table 6.7 reduces the table to the five arcs shown in Table 6.6.

Type 3 Arcs

The derivations for type 3 arcs are analogous to those for type 2 arcs. We present the results here for reference.

For type 3, the simulation of Section 6.7 does not return a complete set of arcs, because the simulation does not take sticking into account. Composing type 3 arcs with a complete set of type 1 arcs makes the type 3 arcs complete. Suppose that α_1 is a type 1 arc, and α_3 a type 3 arc. Denote the cpositions of α_1 by C_1, and the targets by T_1. Denote the cpositions of α_3 by C_3, the spositions by S_3, the sforces by F_3, and the targets by T_3. The composition of α_1 and α_3 produces the two type 3 arcs shown in Table 6.8.

We now have a complete set of type 3 arcs. We need to make this set nonoverlapping, while maintaining completeness, as per Theorem 6.1. Let α_1 and α_2 be a complete pair of type 3 arcs with respect to an atom A. Their composition produces seven arcs. The first arc is the intersection of α_1 and α_2; the respective cpositions, spositions, and sforces of α_1 and α_2 are intersected, and the targets are unioned. The second, third, and fourth arcs result from modifying α_1 to avoid the targets of α_2. The second arc avoids the targets of α_2 by clipping cpositions which are in

Arc	*Cpositions*	*Spositions*	*Sforces*	*Target*
$\alpha_1\alpha_2$	$C_1 \bigcap C_2$	$S_1 \bigcap S_2$	$F_1 \bigcap F_2$	$T_1 \bigcup T_2$
α_1^{c-}	$C_1 - C_2$	S_1	F_1	T_1
α_1^{s-}	C_1	$S_1 - S_2$	F_1	T_1
α_1^{f-}	C_1	S_1	$F_1 - F_2$	T_1
α_2^{c-}	$C_2 - C_1$	S_2	F_2	T_2
α_2^{s-}	C_2	$S_2 - S_1$	F_2	T_2
α_2^{f-}	C_2	S_2	$F_2 - F_1$	T_2

Table 6.9: Composition of a pair of type 3 arcs $\alpha_1 = (C_1, S_1, F_1, T_1)$ and $\alpha_2 = (C_2, S_2, F_2, T_2)$.

α_2. The third arc avoids the targets of α_2 by clipping spositions which are in α_2. The fourth arc avoids the targets of α_2 by clipping sforces which are in α_2. The targets of the second, third, and fourth arcs are the unmodified targets of α_1. The fifth, sixth, and seventh arcs are analogs of the second, third, and fourth arcs, for the case where α_2 is modified to avoid the targets of α_1. The targets of the fifth, sixth, and seventh arcs are the unmodified targets of α_2. The seven arcs are shown in Figure 6.9.

The above intuition can be verified using a more analytical approach. Denote the cpositions of α_1 by C_1, the spositions by S_1, the sforces by F_1, and the targets by T_1. Denote the cpositions of α_2 by C_2, the spositions by S_2, the sforces by F_2, and the targets by T_2. The commanded motions of a type 3 arc are given by triples of the form (c, s, f), where c is a cposition, s is an sposition, and f is an sforce. c can come from one of three nonoverlapping sets, $C_1 \bigcap C_2$, $C_1 - C_2$, and $C_2 - C_1$. s can come from one of three nonoverlapping sets, $S_1 \bigcap S_2$, $S_1 - S_2$, and $S_2 - S_1$. f can come from one of three nonoverlapping sets, $F_1 \bigcap F_2$, $F_1 - F_2$, and $F_2 - F_1$. Crossing these three sets produces 27 possible set triples. For a pair of type 3 motions to be equal, they must have the same cposition, sposition, and sforce. These 27 set triples are mutually nonoverlapping, since they were generated by crossing three sets of nonoverlapping sets. 12 of the set triples do not contain motions that are in either α_1 or α_2:

$$(C_1 - C_2, S_2 - S_1, F_1 \bigcap F_2)$$
$$(C_1 - C_2, S_2 - S_1, F_1 - F_2)$$
$$(C_1 - C_2, S_2 - S_1, F_2 - F_1)$$
$$(C_2 - C_1, S_1 - S_2, F_1 \bigcap F_2)$$
$$(C_2 - C_1, S_1 - S_2, F_1 - F_2)$$
$$(C_2 - C_1, S_1 - S_2, F_2 - F_1)$$

Arc	*Cpositions*	*Spositions*	*Sforces*	*Target*
$\alpha_1\alpha_2$	$C_1 \bigcap C_2$	$S_1 \bigcap S_2$	$F_1 \bigcap F_2$	$T_1 \bigcup T_2$
α_1^-	$C_1 - C_2$	$S_1 - S_2$	$F_1 - F_2$	T_1
α_1^{c-}	$C_1 - C_2$	$S_1 \bigcap S_2$	$F_1 \bigcap F_2$	T_1
α_1^{s-}	$C_1 \bigcap C_2$	$S_1 - S_2$	$F_1 \bigcap F_2$	T_1
α_1^{f-}	$C_1 \bigcap C_2$	$S_1 \bigcap S_2$	$F_1 - F_2$	T_1
α_1^{cs-}	$C_1 - C_2$	$S_1 - S_2$	$F_1 \bigcap F_2$	T_1
α_1^{cf-}	$C_1 - C_2$	$S_1 \bigcap S_2$	$F_1 - F_2$	T_1
α_1^{sf-}	$C_1 \bigcap C_2$	$S_1 - S_2$	$F_1 - F_2$	T_1
α_2^-	$C_2 - C_1$	$S_2 - S_1$	$F_2 - F_1$	T_2
α_2^{c-}	$C_2 - C_1$	$S_1 \bigcap S_2$	$F_1 \bigcap F_2$	T_2
α_2^{s-}	$C_1 \bigcap C_2$	$S_2 - S_1$	$F_1 \bigcap F_2$	T_2
α_2^{f-}	$C_1 \bigcap C_2$	$S_1 \bigcap S_2$	$F_2 - F_1$	T_2
α_2^{cs-}	$C_2 - C_1$	$S_2 - S_1$	$F_1 \bigcap F_2$	T_2
α_2^{cf-}	$C_2 - C_1$	$S_1 \bigcap S_2$	$F_2 - F_1$	T_2
α_2^{sf-}	$C_1 \bigcap C_2$	$S_2 - S_1$	$F_2 - F_1$	T_2

Table 6.10: Composition of a pair of type 3 arcs $\alpha_1 = (C_1, S_1, F_1, T_1)$ and $\alpha_2 = (C_2, S_2, F_2, T_2)$.

$$(C_1 - C_2, S_1 \bigcap S_2, F_2 - F_1)$$
$$(C_1 - C_2, S_1 - S_2, F_2 - F_1)$$
$$(C_2 - C_1, S_1 \bigcap S_2, F_1 - F_2)$$
$$(C_2 - C_1, S_2 - S_1, F_1 - F_2)$$
$$(C_1 \bigcap C_2, S_1 - S_2, F_2 - F_1)$$
$$(C_1 \bigcap C_2, S_2 - S_1, F_1 - F_2)$$

Thus, there are fifteen feasible triples of motion sets. These triples generate the fifteen arcs shown in Table 6.10. These arcs are complete, because we generated all feasible motion set triples from α_1 and α_2, and we carried with them all possible targets. They are reliable because they are complete and nonoverlapping.

When two type 3 arcs α_1 and α_2 have identical cpositions, spositions, and targets, they can be merged into a single type 3 arc whose sforces are the union of the sforces of α_1 and α_2. When two type 3 arcs α_1 and α_2 have identical cpositions, sforces, and targets, they can be merged into a single type 3 arc whose spositions are the union of the spositions of α_1 and α_2. When two type 3 arcs α_1 and α_2 have identical spositions, sforces, and targets, they can be merged into a single type 3 arc whose cpositions are the union of the cpositions of α_1 and α_2. Iteratively applying these simplifications to Table 6.10 reduces the table to the seven arcs shown in Table 6.9.

Expanding the Arcs of a State with Multiple Atoms

We have presented a method for computing the strong arcs of a singleton state. If a state X consists of multiple atoms, then we can compute the strong arcs of each atom in X individually, as if each atom were a singleton state. Composing all of the resulting arcs together will then make them reliable with respect to X.

Removing Free Space Targets

We now have a state X, and a set Ψ of arcs that contains all reliable motions that are possible from X. Recall that in our atom decomposition, an atom was reserved for free space. Its purpose was essentially to prevent motions from terminating in free space. In our implementation, we do not actually do this. Instead, we treat free space as a special case. We construct a polyhedral volume which represents free space. We then grow this volume by ϵ_p. The grown volume represents all positions that may cause the robot to terminate in free space, under the given position uncertainty. Finally, we clip this volume out of the commanded positions of each arc of X.

Installing the Arcs in the State Graph

We now have a set of strong arcs from X, which must be installed in the state graph. The origin of each arc α is at X. The heads of α point at the primitive states that can be formed from the target atoms of α, using the algorithm described in Section 6.4.1.

Summary

This subsection described a method for constructing the arcs of a state X. We began by computing the set of motions that were possible from each atom A of X. For each A, the motions were grouped into weak arcs whose motions share the same target. The arcs of X were then composed together, producing reliable strong arcs. Motions which might terminate in free space were then removed from the arcs. Finally, each arc was installed in the state graph, with its origin at X and pointing at primitive states composed of target atoms from the arc.

6.7 Simulation

The previous section assumed the existence of a simulation which computes the set of all motions that are possible from a given atom. This section gives the details of the simulation. Given an atom A, the simulation returns a set of weak arcs. The motions of each weak arc share the same target when starting from A.

From A, the robot can do one of two things. First, it can terminate in A without ever leaving it. Simulation of this case is described in Section 6.7.1. Termination can be accomplished by some combination of sticking (Section 6.7.4), position sensing (Section 6.7.6), and force sensing (Section 6.7.7). Alternatively, the robot can proceed directly to another atom B, and continue on from there. Simulation of this case is described in Section 6.7.2. This case requires that we simulate the direct reaching of one atom from another. Direct reaching is discussed in Section 6.7.5.

6.7.1 Termination in A

To represent termination in A, we will construct three arcs, one of each arc type. The target of each arc is A.

- The first arc is a type 1 arc. Section 6.7.4 presents a method for computing the set of commanded positions that may cause the robot to stick in an atom if it reaches it. These positions become the cpositions of our arc.
- The second arc is a type 2 arc. Since the robot is already in A, any commanded position will cause it to reach A. Thus, the cpositions of the arc are set to all possible commanded positions. Section 6.7.6 presents a method for computing the set of sensed positions that are consistent with an atom. These sensed positions become the spositions of our arc.
- The third arc is a type 3 arc. Since the robot is already in A, any commanded position will cause it to reach A. Thus, the cpositions of the arc are set to all possible commanded positions. Section 6.7.6 presents a method for computing the set of sensed positions that are consistent with an atom. These sensed positions become the spositions of our arc. Section 6.7.7 presents a method for computing the set of sensed forces that are consistent with an atom. These sensed forces become the sforces of our arc.

6.7.2 Termination in Other Atoms

If the robot is to terminate in another atom, then it must leave A, and proceed directly to some other atom B. Once it has reached B, it can either terminate in B, or go somewhere else.

To leave A, the robot must not stick in A. We can compute the set S_A of commanded positions which are guaranteed to stick in A. A method to do this is described in Section 6.7.4. These commanded positions should be avoided to allow the possibility of leaving A.

We then enumerate each atom $B \neq A$ which is a candidate for direct reachability. The candidate set should include all faces in the environment, and any edges and vertices that are connected to A. Isolated edges and vertices can be ignored, since the probability of reaching them from A is zero. Note that for two atoms to be connected, one of them must bound the other.

For each B, we compute the set of commanded positions C_B that might cause the robot to reach B directly from A. Section 6.7.5 presents a method to do this. We then clip S_A from C_B, to remove trajectories that are guaranteed to stick in A. We then recursively call the simulation to compute the set of arcs representing the possible motions from B. For each arc α_B in this set, we intersect the cpositions of

α_B with C_B. This ensures that the target of α_B is reachable from A. We collect the arcs for each B, until all reachable atoms have been processed.

The recursion ends when either there are no atoms that are reachable from A, or A is active in the chain of atoms that are currently being expanded. If A is active, then the caller is put on a waitlist for the results of A. A waitlist entry consists of the following fields:

waiter The atom that is waiting.

waitee The atom that is being waited for.

cpositions The set of commanded positions that may cause the robot to reach the waitee from the waiter.

A waitlist entry cannot be processed until the waitee is finished. For this reason, the simulation returns a waitlist a well as a set of arcs.

When the simulation of A calls for a recursive simulation of B, the recursive call returns a waitlist. On return, the simulation of A checks through this waitlist. If an entry is found with B waiting for an atom $J \neq A$, then the simulation creates a new waitlist entry with A waiting for J as well. The cpositions of A's waitlist entry are equal to C_B intersected with the cpositions of B's waitlist entry.

When the simulation of A completes its computations, it checks through the waitlist for atoms that are waiting for A. If one is found, it is removed from the list. A new set of arcs is created by copying the arcs of A, and intersecting their cpositions with the cpositions of the waitlist entry. This ensures that the targets of the new arcs are reachable by the waiter. The new arcs are then appended to the previous arcs of the waiter.

If A is on the waitlist when the simulation of A completes, then any atom waiting for A must also wait for atoms that A is waiting for. The simulation checks through the waitlist for an atom B that is waiting for A. If such a B is found, and there is another waitlist entry in which A is waiting for an atom $J \neq B$, then a new waitlist entry is created in which B is waiting for J. The cpositions of the new waitlist entry are equal to the intersection of the cpositions of the other two waitlist entries.

6.7.3 Memoization

We can save the results of A's simulation in a lookup table, in the event that A is to be simulated again. This can happen when A is reachable from another atom in the current state expansion, or when A is needed to expand another state.

Note that in the expansion of an atom A, the simulation calls itself recursively on each atom B that is directly or indirectly reachable from A. Thus, after one state has been expanded, it is likely that most of the atoms in an environment have been expanded. This means that with memoization, state expansions beyond the first can usually be performed cheaply.

There is one minor difficulty having to do with waitlists. The simulation of an atom A may have completed but left A on a waitlist. When the results of the simulation are found in the lookup table by a subsequent call, all currently existing waitlists must also be checked, to see if A is waiting for anything. Any waitlist entries found must be returned along with the arcs in the lookup table.

The overall structure of the simulation has now been described. The rest of the section concerns itself with the details of subtasks.

6.7.4 Sticking

The subtask here is to compute the set of commanded positions that cause sticking on an atom A if the robot reaches it. There are two types of sticking calculations. The *weak sticking region* of A consists of the commanded positions that might cause sticking on A if the robot reaches it. The *strong sticking region* of A consists of the commanded positions that are guaranteed to cause sticking on A if the robot reaches it.

Let's start with a simple case which demonstrates the basic principles of weak and strong sticking. Suppose that A is a point on a face. The friction cone of A is depicted in Figure 2.3. If we negate A's friction cone, then we have the set of robot forces that will cause sticking on A. In the absence of control and sensing error, a generalized spring command causes the robot to impart a force in the direction of the commanded position. Thus, the negative friction cone also represents the set of commanded positions that will cause sticking on A, in the absence of control and sensing uncertainty.

Under velocity uncertainty, the force imparted by the robot may err by an angle that may be as large as θ_v. This is where weak and strong sticking come into play. To represent the weak sticking region of A, we need to increase the angle of the negative friction cone by θ_v. We call the resulting cone the *weak sticking cone* of A. This cone includes commanded positions which may cause the robot to stick due to velocity error. Figure 6.18 shows the weak sticking cone of A.

To represent the strong sticking region of A, we need to decrease the angle of the negative friction cone by θ_v. The resulting cone is the *strong sticking cone* of A. This cone contains commanded positions which are guaranteed to cause sticking, even under maximal velocity error. Figure 6.19 shows the strong sticking cone of A.

Under position sensing uncertainty, the robot may misjudge its position by as much as ϵ_s. This may cause the robot to aim for a commanded position which is off by as much as ϵ_s. This will alter the force imparted by the robot. We can take this into account by growing the weak sticking cone volumetrically by ϵ_s, and shrinking the strong sticking cone volumetrically by ϵ_s.

We have finished the case of a point on a face. The atoms that our planner deals with are (a) entire faces, (b) edges that are at the intersection of two faces, and (c) vertices that are at the intersection of a number of faces. These cases are detailed below.

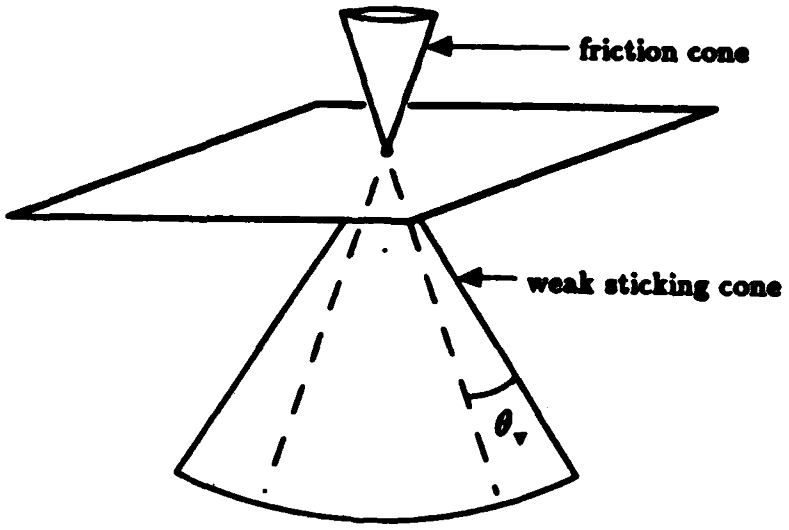


Figure 6.18: The weak sticking cone of a point on a face, under velocity uncertainty.

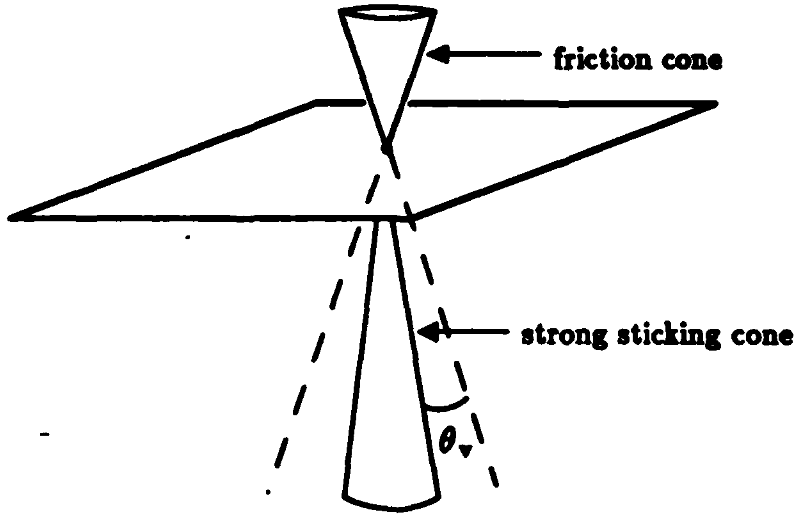


Figure 6.19: The strong sticking cone of a point on a face, under velocity uncertainty.

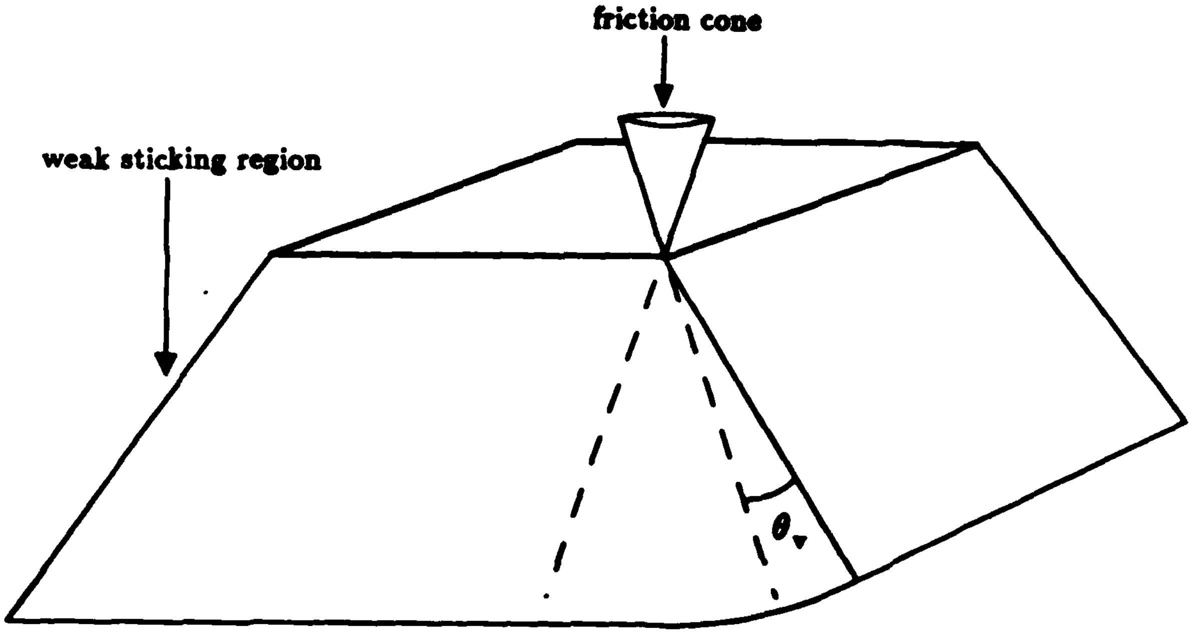


Figure 6.20: The weak sticking region of a face is equal to the union of the weak sticking cones of the points on the face.

Sticking on a Face

Let F be a face, and ϕ its friction cone angle. We are to compute a set P of commanded positions that cause sticking on A.

We begin with weak sticking. To compute the weak sticking region of a face, we could compute the weak sticking cone for each point $\mathbf{x} \in f$, and union the resulting cones together. See Figure 6.20. With this as intuition, we will present a more efficient algorithm for computing P. In most cases, P will be a convex set. The algorithm computes P by constructing its planar bounds.

Assume initially that $\phi + \theta_v \leq \frac{\pi}{2}$. In order that the robot stay on F, commanded positions must be interior to F. For weak sticking, we bound P by a plane which is parallel to F, and exterior to it by ϵ_s. This region includes commanded positions that might be interpreted as interior to F under position sensing error.

The remaining bounds on P are generated from the bounding edges of F. For each edge $\bar{e}$, a bounding plane is constructed that contains $\bar{e}$. To account for velocity uncertainty, the plane makes an angle of $\phi + \theta_v$ with an interior normal vector to F, as shown in Figure 6.21. To account for position sensing uncertainty, the plane is then translated exterior to $\bar{e}$ by ϵ_s.

Figure 6.22 shows the weak sticking region of the bottom of a hole, as computed by the planner.

For the case where $\phi + \theta_v > \frac{\pi}{2}$, it is possible to command a position that is exterior to F, and have the robot stick on F. In this case, P will have a concavity. For simplicity, we will skip the details of this construction.

The set P is always convex for strong sticking. In order that commanded posi-

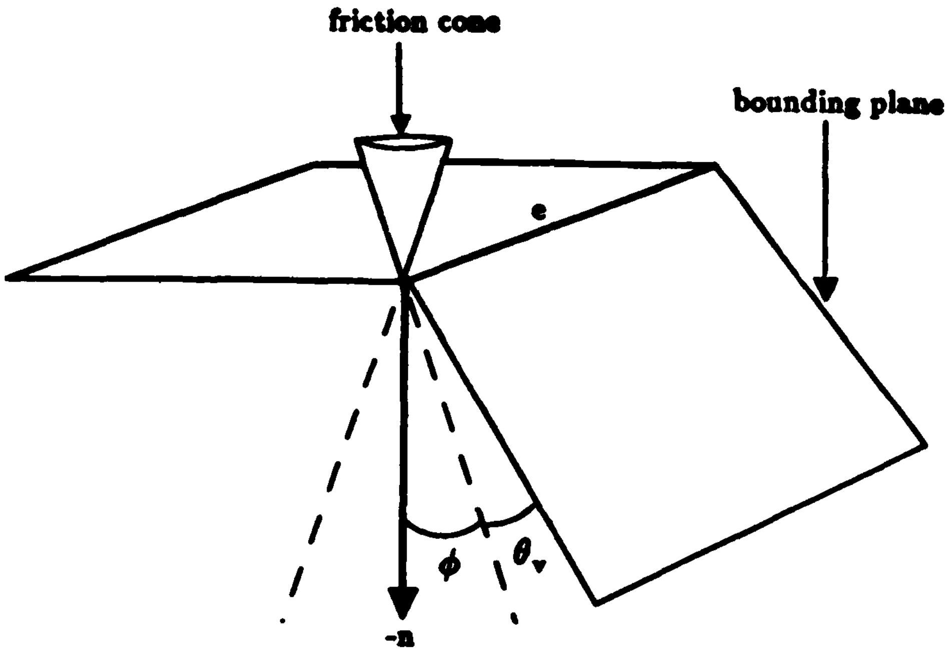


Figure 6.21: Each edge $\bar{e}$ of a face generates a bounding plane on the weak sticking positions of the face. ϕ is the friction cone angle, and θ_v is the velocity uncertainty angle.

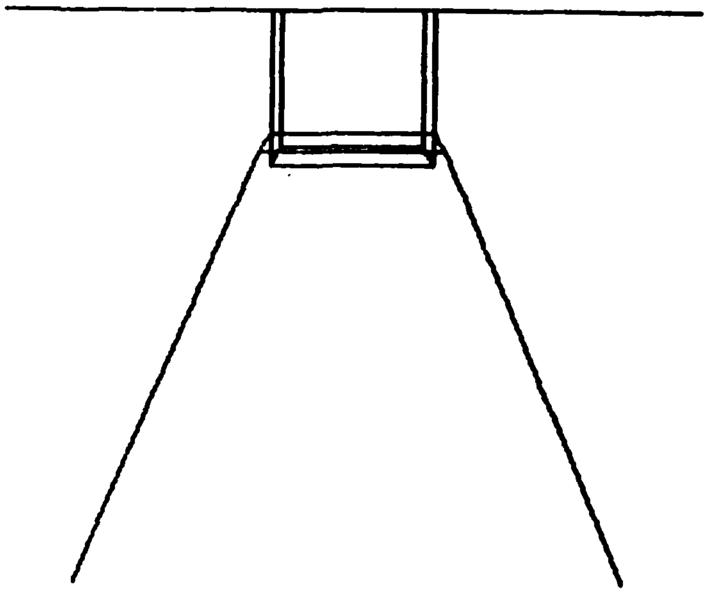


Figure 6.22: The weak sticking region of the bottom of a hole, as computed by the planner.

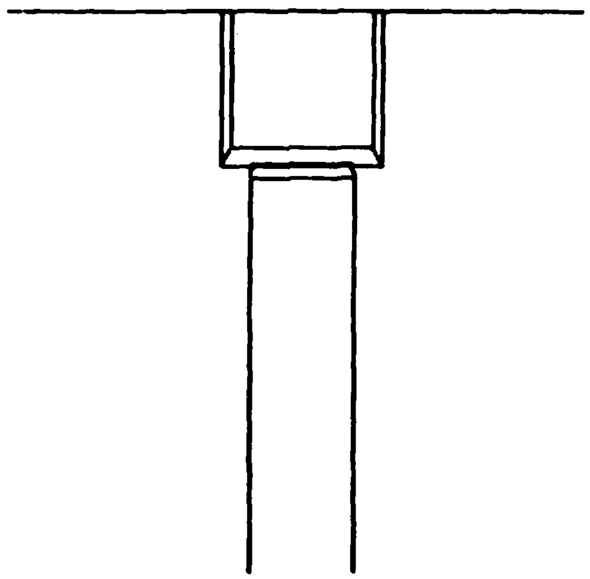


Figure 6.23: The strong sticking region of the bottom of a hole, as computed by the planner.

tions be interior to F, we bound P by a plane which is parallel to F and interior to it by ϵ_s. This region contains commanded positions that are guaranteed to be interpreted as interior to F, even under maximal position sensing error.

Initially, one might suspect that strong sticking on F would require intersecting the strong sticking cones for each $\mathbf{x} \in f$. This would ensure that no matter where the robot strikes F, the commanded position will cause it to stick there. However, this intersection will produce an empty set of commanded positions in most cases.

Fortunately, it is not necessary that the robot stick at the point where it strikes F. It is acceptable for the robot to slide to another point on F and stick there, as long as it does not fall off F along the way. When the robot slides on F, it slides toward the projection of the commanded position onto F. Sliding trajectories stay within a planar cylinder of radius ϵ_t, as shown in Figure 3.14. To avoid falling off F, commanded positions must be at least ϵ_t inside the bounding edges of F. To implement this constraint, we bound P for each edge $\bar{e}$ by a plane which is perpendicular to F, parallel to $\bar{e}$, and interior to $\bar{e}$ by ϵ_t. Figure 6.23 shows the strong sticking region of the bottom of a hole, as computed by the planner.

Sticking on an Edge

Let $\bar{e}$ be an edge, cobounded by the faces f_1 and f_2. We will compute the set of commanded positions that cause sticking on $\bar{e}$.

We begin with weak sticking. For each point $\mathbf{x} \in \bar{e}$, we could compute the weak sticking cone of $\mathbf{x}$, and union all of the cones together. The actual algorithm merely constructs the bounding planes of this set.

Figure 2.4 shows the friction cone for a point on a convex edge. The weak

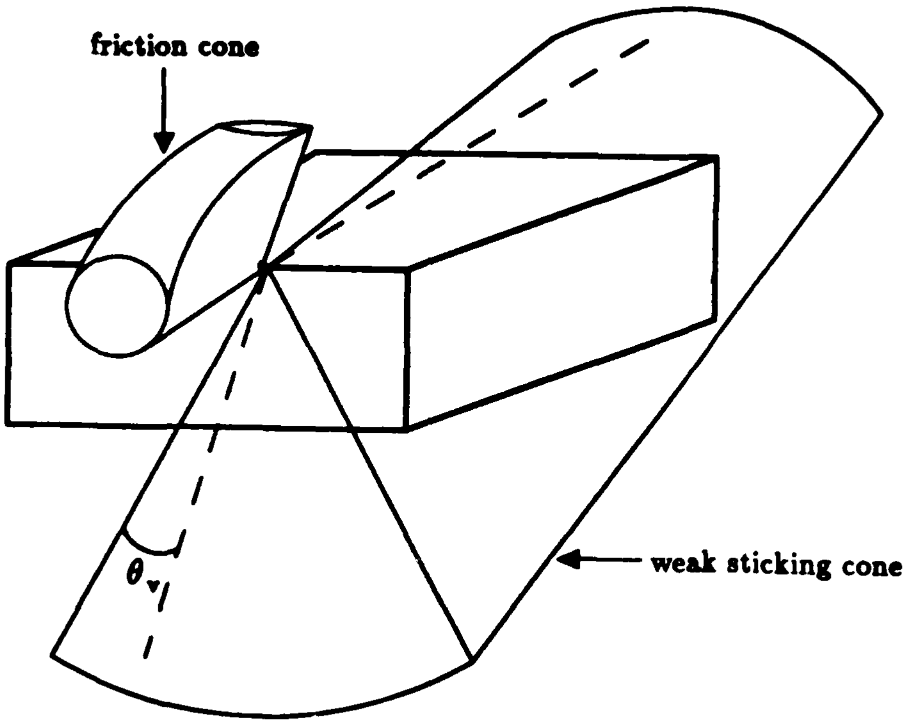


Figure 6.24: The weak sticking cone of a point on a convex edge.

sticking cone of $\mathbf{x}$ is equal to the negative friction cone, with the cone angle increased by θ_v, and the cone then grown volumetrically by ϵ_s, to account for velocity and position sensing uncertainty. Note that the cone angle of a nonsymmetric cone can be increased or decreased by rotating each bounding ray of the cone about a perpendicular tangent at the vertex. Figure 6.24 shows the weak sticking cone of a point on a convex edge. The weak sticking region of $\bar{e}$ is equal to the union of all of the weak sticking cones. The weak sticking region of a convex edge is shown in Figure 6.25.

The weak sticking region of an edge takes a radically different shape when the maximum friction cone angle plus θ_v exceeds $\frac{\pi}{2}$. It is now possible to command a position that is exterior to both f_1 and f_2, and have the robot stick on $\bar{e}$. In this case, the weak sticking region will have a concavity. For simplicity we will skip the details of this construction.

One might initially suspect that strong sticking on $\bar{e}$ would require intersecting the strong sticking cones for each $\mathbf{x} \in \bar{e}$. This would ensure that no matter where the robot strikes $\bar{e}$, the commanded position will cause it to stick there. However, this intersection would produce an empty set of commanded positions in most cases. Fortunately, it is not necessary that the robot stick at the point where it strikes $\bar{e}$. It is acceptable for the robot to slide to another point on $\bar{e}$ and stick there, as long as it does not fall off $\bar{e}$ along the way. It is possible to compute strong sticking on $\bar{e}$ by unioning the strong sticking cones of each $\mathbf{x} \in \bar{e}$. Suppose that the commanded

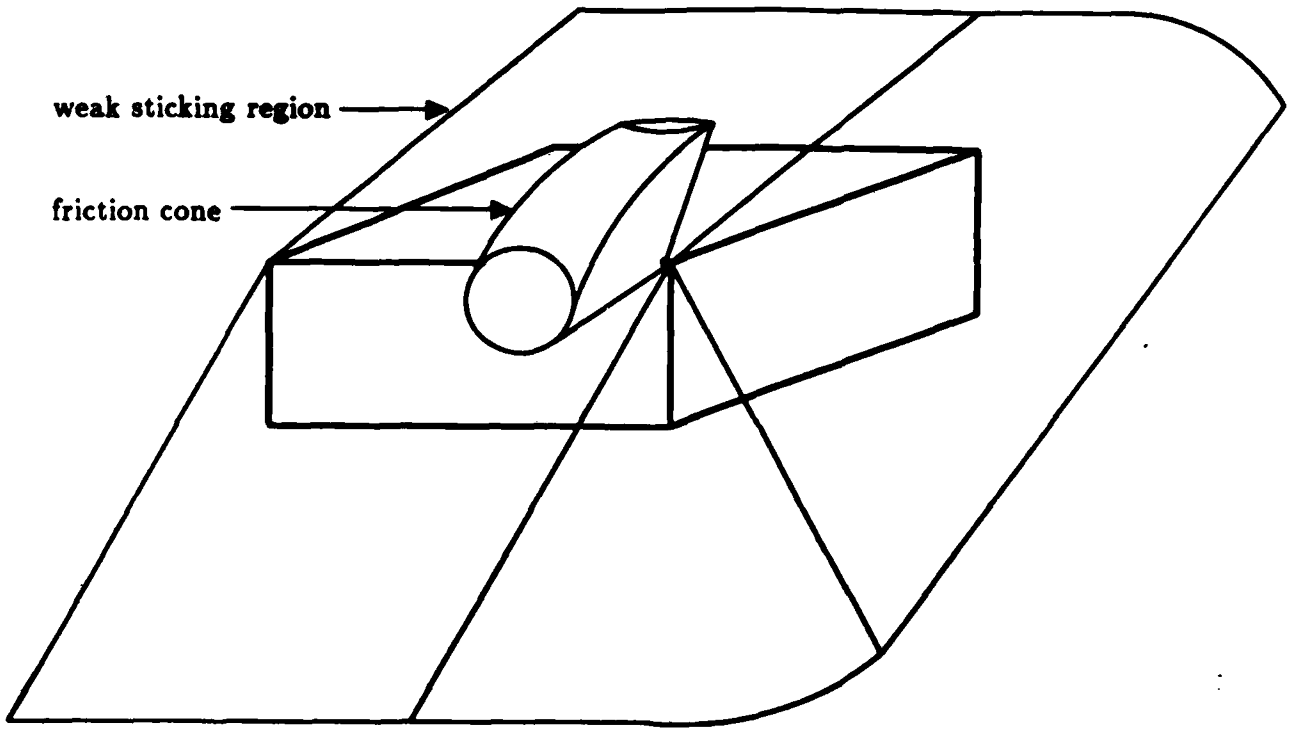


Figure 6.25: The weak sticking region of a convex edge.

position **p** is in the union of the strong sticking cones. If the robot strikes $\bar{e}$ at some point **x**, and **p** is not in the strong sticking cone of **x**, then the edge imparts the reaction force shown in Figure 6.26. The resulting force on the robot causes the robot to slide along $\bar{e}$, without falling off, until it sticks at the first point **x**′ which contains **p** in its strong sticking cone.

The strong sticking cone of a point $\mathbf{x} \in e$ is equal to the negative friction cone of **x**, with the cone angle decreased by θ_v, and the cone then shrunk by ϵ_s, to account for velocity and position sensing uncertainty. Figure 6.27 shows the strong sticking cone of a point on a convex edge, under velocity uncertainty. The strong sticking region of a convex edge is shown in Figure 6.28.

Sticking on a Vertex

Let **v** be a vertex, which is the intersection of a set of faces. We will compute a set of commanded positions which stick on **v**.

The friction cone of a convex vertex **v** is shown in Figure 2.5. The weak sticking region of **v** is equal to the negative friction cone, with the cone angle increased by θ_v, and the cone then grown volumetrically by ϵ_s, to account for velocity and position sensing uncertainty.

The strong sticking region of **v** is equal to the negative friction cone, with the cone angle decreased by θ_v, and the cone then shrunk volumetrically by ϵ_s, to

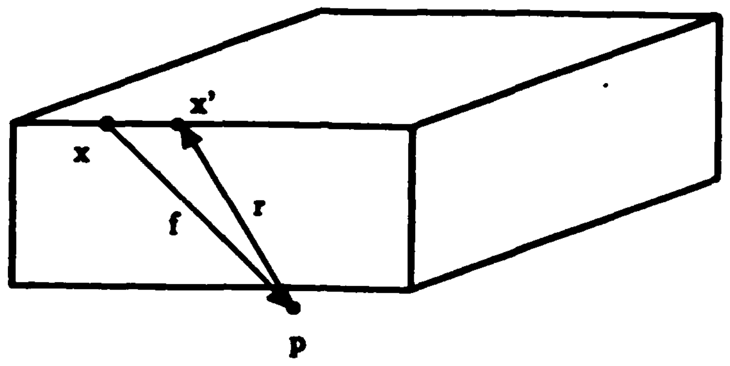


Figure 6.26: A strong sticking point may cause sliding on an edge. The commanded position is $\mathbf{p}$. The robot strikes the edge at $\mathbf{x}$, and imparts the force $\mathbf{f}$. The reaction force of the edge is $\mathbf{r}$.

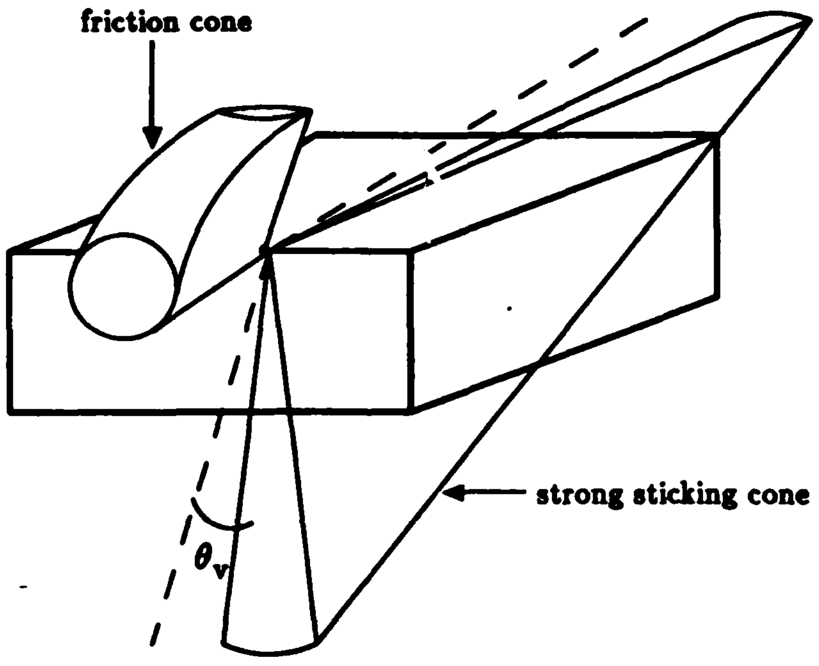


Figure 6.27: The strong sticking cone of a point on a convex edge, under velocity uncertainty.

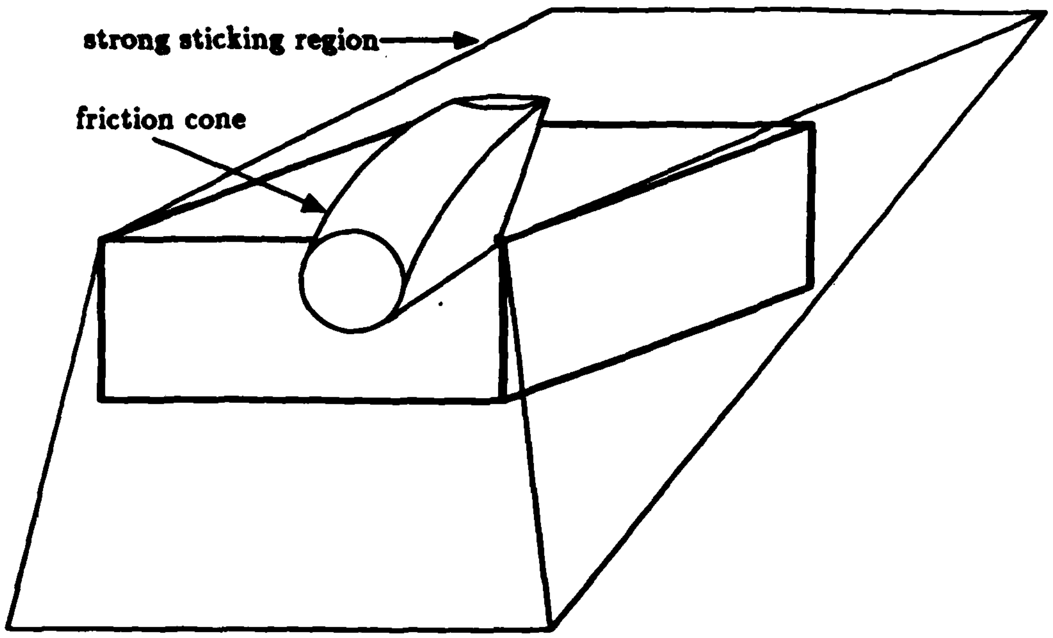


Figure 6.28: The strong sticking region of a convex edge, under velocity uncertainty.

account for velocity and position sensing uncertainty.

6.7.5 Direct Reaching

Our task here is to compute the commanded positions that cause the robot to reach an atom B directly from an atom A. There are two types of reaching calculations. *Weak reaching positions* are commanded positions that might cause the robot to reach B directly from A. *Strong reaching positions* are commanded positions that guarantee reaching B directly from A.

Suppose that we have a function to compute the set of commanded positions that reach B directly from A, ignoring obstacles in between. Then we can make the following observations:

- The commanded positions which weakly reach B directly from A are equal to the commanded positions which weakly reach B directly from A, ignoring obstacles, minus the commanded positions which strongly reach O directly from A, ignoring obstacles, for all atoms O between A and B.
- The commanded positions which strongly reach B directly from A are equal to the commanded positions which strongly reach B directly from A, ignoring obstacles, minus the commanded positions which weakly reach O directly from A, ignoring obstacles, for all atoms O between A and B.

The first observation says that to reach B directly and weakly, the robot must choose commanded positions which might strike B, and aren't guaranteed to strike

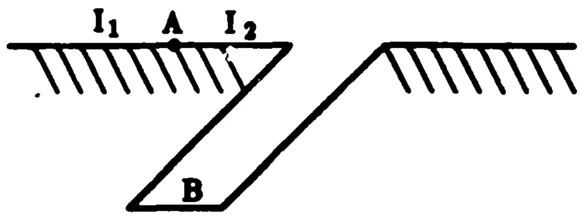


Figure 6.29: We are to compute the weak reaching positions of B directly from A.

some obstacle between A and B. The second observation says that to reach B directly and strongly, the robot must choose commanded positions which will strike B, and can't possibly strike some obstacle between A and B.

These observations lead to a procedure for computing direct reaching positions. The procedure calls two basic subfunctions. The first subfunction computes obstacles that are between A and B. The second subfunction computes commanded positions which reach B directly from A, ignoring obstacles. Both of these subfunctions are described below.

An interesting bug arises in the case of weak reaching. Consider the two-dimensional example shown in Figure 6.29. A is a point on the table, and B is an edge at the bottom of a slanted hole. A connects two edges I_1 and I_2 above the hole. We are to compute the weak reaching positions of B directly from A. Figure 6.30 shows the weak reaching positions of B directly from A, ignoring obstacles. Figure 6.31 shows the strong reaching positions of I_2 directly from A, ignoring obstacles. Figure 6.32 shows the strong reaching positions of I_1 directly from A, ignoring obstacles. Clipping the strong reaching positions of I_1 and I_2 from the weak reaching positions of B yields the commanded positions shown in Figure 6.33. These commanded positions supposedly will allow the robot to sneak between I_1 and I_2, which is impossible. The solution is to clip the weak reaching positions of A from the result. The weak reaching positions of A from itself are equal to all possible commanded positions. After clipping, no weak reaching positions of B remain.

More generally, for weak reaching, if there exists an edge or vertex which connects a set of faces that are all strongly reachable by commanded positions that might reach B, then the above bug may occur. To prevent the bug, we must clip commanded positions which can weakly reach the connecting edge or vertex from the result. These commanded positions are precisely the ones that should not have been clipped by the strong reaching computation.

Computing Obstacles

Here, our task is to compute the obstacle atoms that are in the path between two atoms A and B. Since the robot can veer from its course by as much as ϵ_t, we are looking for obstacles that are in the volume of space between A and B, grown laterally by ϵ_t.

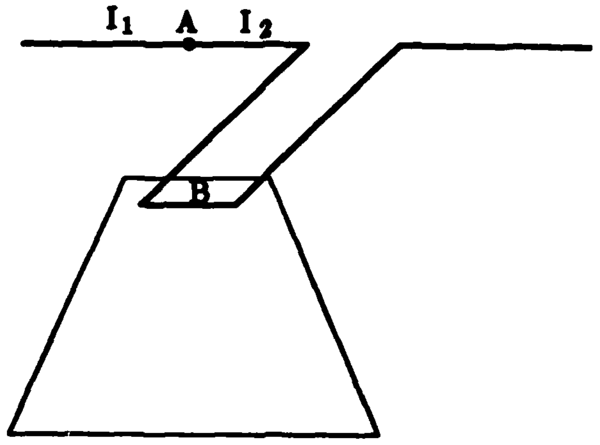


Figure 6.30: The weak reaching positions of B directly from A, ignoring obstacles.

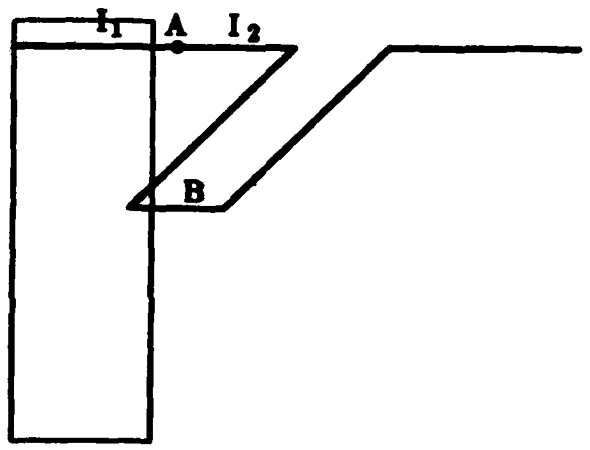


Figure 6.31: The strong reaching positions of I_1 directly from A, ignoring obstacles.

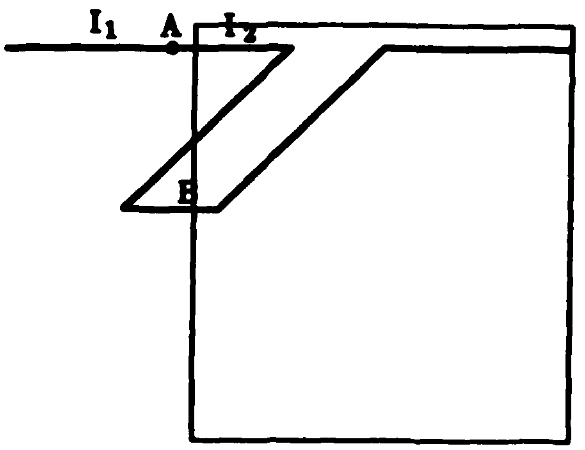


Figure 6.32: The strong reaching positions of I_2 directly from A, ignoring obstacles.

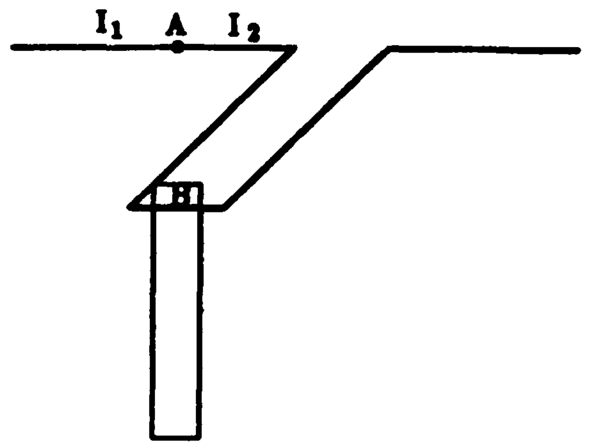


Figure 6.33: The weak reaching positions of B directly from A, ignoring obstacles, minus the strong reaching positions of I_1 and I_2 directly from A, ignoring obstacles.

If A and B are connected, then there are no obstacles between them. If A and B are unconnected but share a common face, then the obstacle that separates them is the face itself.

If A and B do not share a common face, then any atom which is connected to A or B is an obstacle. The remaining obstacles are atoms which are in the volume of space between A and B, grown laterally by ϵ_t. Each atom in the environment which intersects this volume is an obstacle.

Direct Reaching Between Connected Atoms, Ignoring Obstacles

We are to compute the commanded positions that allow the robot to reach an atom B directly from an atom A, ignoring obstacles between A and B. There are two cases to consider. The first case, when A and B are connected, is discussed in this subsection. The case where A and B are unconnected is discussed in the next subsection.

Assume that A and B are connected on a common face F. We will compute a convex polyhedron P of reaching positions.

In order that the robot slide from A to B, it must remain on F. This means that commanded positions must be interior to F. For weak reaching, we bound P by a plane which is parallel to F, and exterior to it by ϵ_s. This region includes commanded positions that might be interpreted as interior to F under position sensing error. For strong reaching, we bound P by a plane which is parallel to F, and interior to it by ϵ_s. This region contains commanded positions that are guaranteed to be interpreted as interior to F, even under maximal position sensing error.

We need a constraint on P to make sure that the robot passes from A to B. To do this, we construct separating planes between A and B. Figure 6.34 shows the placement of the separating planes for various combinations of A and B. Normally,

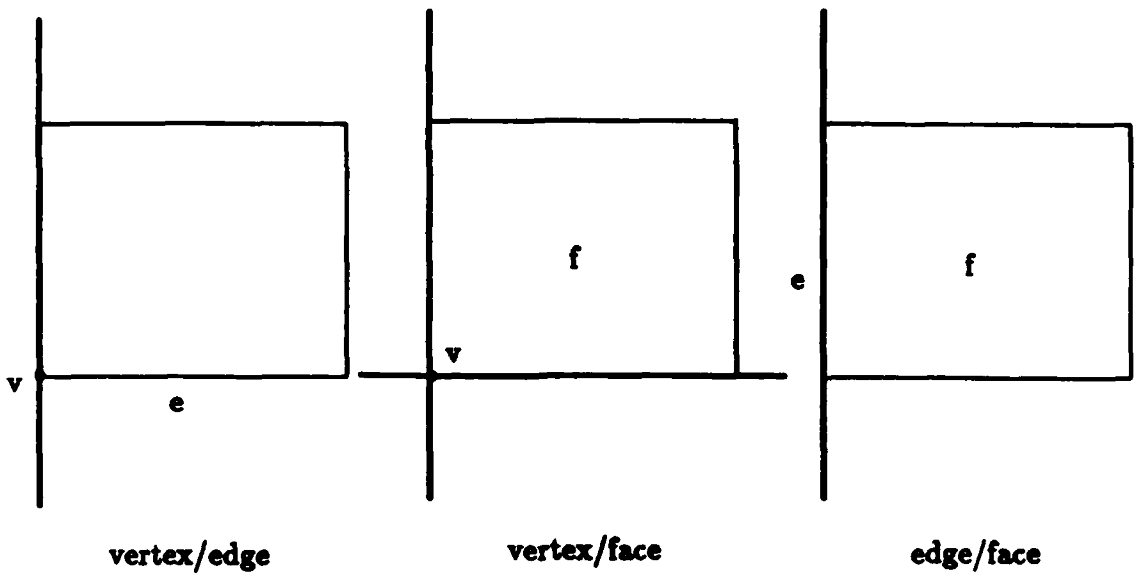


Figure 6.34: The placement of separating planes between two connected atoms, for various combinations.

a single plane will do. If A is a vertex, and B is a face, or vice versa, then two separating planes are needed, one for each edge that the vertex connects.

For each separating plane, we construct a bound on P to ensure that the robot crosses the plane. For weak reaching, we move the separating plane toward A by ϵ_t, and bound P by it. Commanded positions must be on the B side of the plane. This bound includes positions which might cause B to be reached due to trajectory error. For strong reaching, we move the separating plane toward B by ϵ_p, and bound P by it. This bound ensures reaching B even under maximal positioning error.

If A is a face, and B is a bounding edge, then two other planar bounds on P are needed for strong reaching. Each bound is orthogonal to A, parallel to an edge of A which is adjacent to B, and interior to the edge by ϵ_t. Figure 6.35 shows an example. These additional bounds ensure that trajectories do not slide off A on the way to B.

If B is a vertex, then strong reaching is impossible, since a point cannot be reached accurately under control uncertainty. Similarly, if A is a vertex, and B is a cobounding edge, then strong reaching is impossible.

Direct Reaching Between Unconnected Atoms, Ignoring Obstacles

Now we are to compute the commanded positions that allow the robot to reach an atom B directly from an unconnected atom A, ignoring obstacles between A and B.

To simplify the presentation, we will begin by describing the computation in two dimensions. Assume that A and B are either vertices or edges in the plane. We will construct a convex polygon P that contains the reaching positions of B from

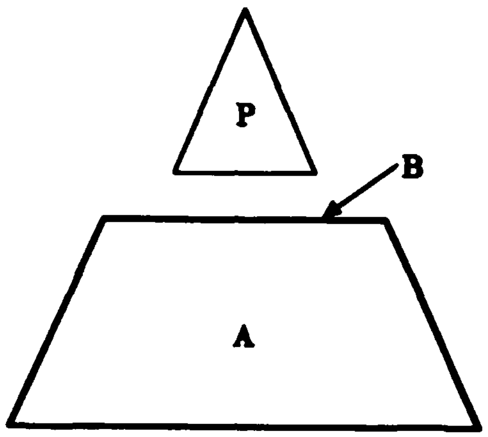


Figure 6.35: P contains the strong reaching positions of the edge B from its cobounding face A. Linear bounds on P are derived from B and from its adjacent edges.

A. Keep in mind that an edge in the plane is the counterpart to a face in three dimensions. Thus, an edge has a unique normal vector, which defines a half space which is exterior to it.

First, we will treat the case where A and B face each other. For this to be true, A must be exterior to B, and B exterior to A. To ensure this, we must compute a normal vector for both. If A or B is an edge, then its normal vector is given. If A is a vertex, and B is an edge, then A is assigned the reverse of B's normal vector. If B is a vertex, and A is an edge, then B is assigned the reverse of A's normal vector. If both A and B are vertices, then their normal vectors point at each other. To ensure that A and B face each other, we clip from A the portion that is interior to B, and from B the portion that is interior to A. We continue on, using what remains of the two.

Commanded positions must be behind B. To enforce this constraint, we bound P by an edge which is perpendicular to B's normal vector. For weak reaching, we place this bound in front of B by ϵ_t, to allow reaching B due to trajectory error. For strong reaching, we place this bound behind B by ϵ_p, to ensure reaching B despite positioning error.

For weak reaching, for each $\mathbf{p} \in P$, a nominal trajectory of the robot for some $\mathbf{x} \in A$ must come within ϵ_t of B at some point. This allows for trajectories that strike B due to trajectory error. We need to construct bounds on P such that this property holds. Figure 6.36 shows the case where A and B are both vertices. Figure 6.37 shows the case where A is a vertex and B is an edge. Figure 6.38 shows the case where A is an edge and B is a vertex. Figure 6.39 shows the case where A and B are both edges. To construct one of these bounds, we construct a line which connects an extreme point of A with the extreme point of B on the opposite side. The line is then rotated about the extreme point of A such that it is exterior to B, and its perpendicular distance from the extreme point of B is ϵ_t. The rotated line bounds P.

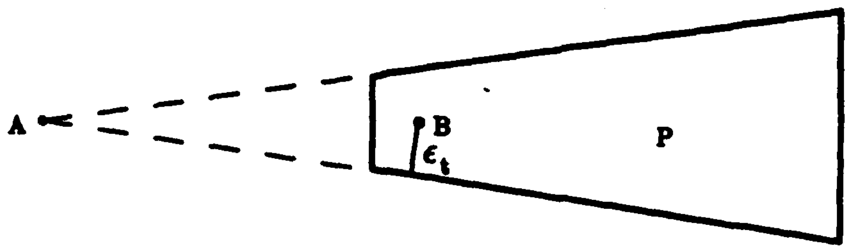


Figure 6.36: The weak reaching positions for the case where *A* and *B* are both vertices.

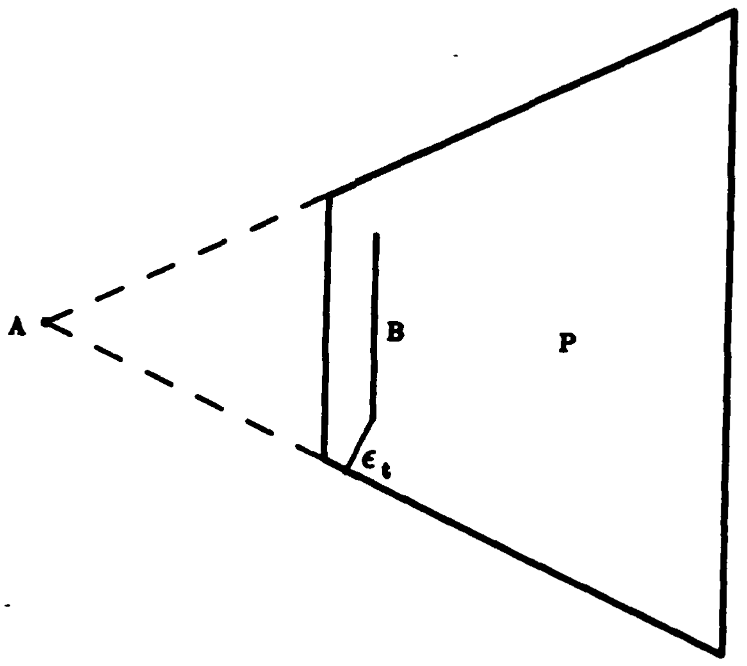


Figure 6.37: The weak reaching positions for the case where *A* is a vertex, and *B* is an edge.

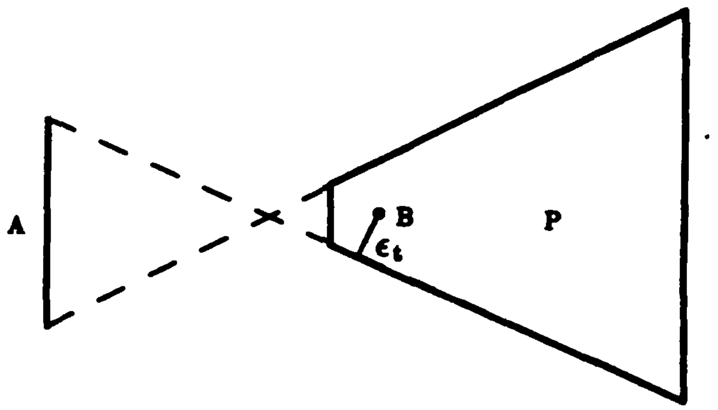


Figure 6.38: The weak reaching positions for the case where A is an edge, and B is a vertex.

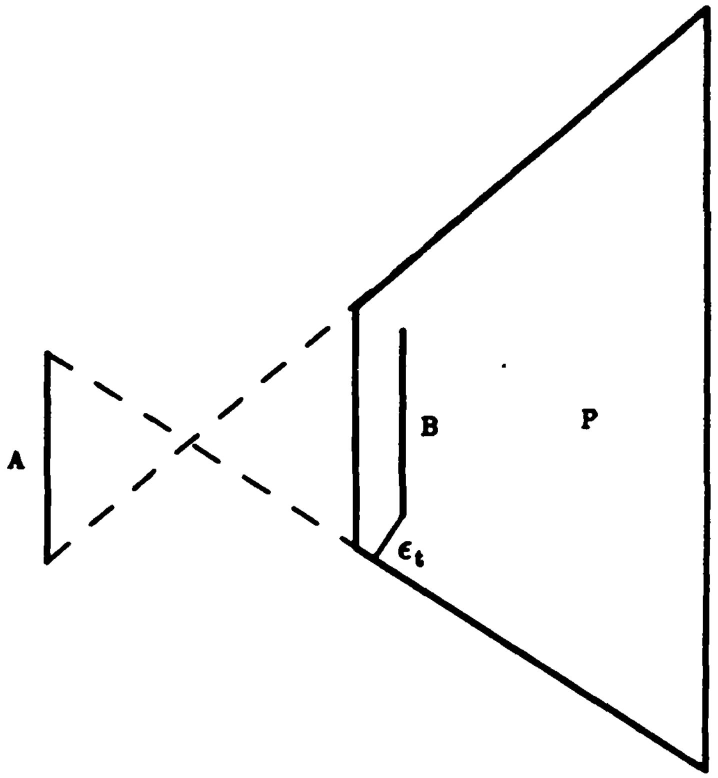


Figure 6.39: The weak reaching positions for the case where A and B are both edges.

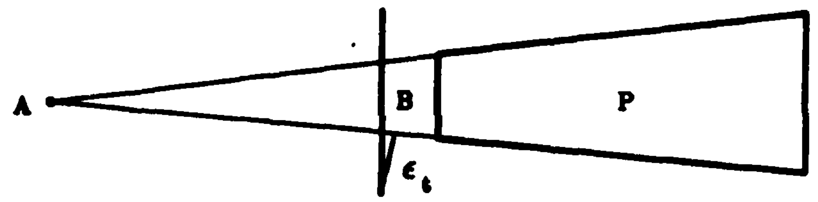


Figure 6.40: The strong reaching positions for the case where A is a vertex, and B is an edge.

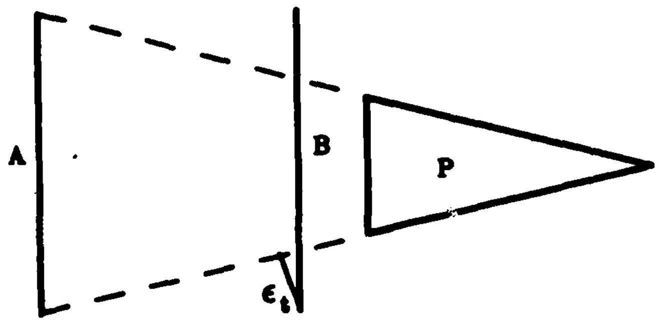


Figure 6.41: The strong reaching positions for the case where A and B are both edges.

For strong reaching, for each $\mathbf{p} \in P$, all nominal trajectories of the robot must be interior to B by at least ϵ_t when they reach it. This ensures that they won't pass it by due to trajectory error. We need to construct bounds on P such that this property holds. Due to control uncertainty, strongly reaching a vertex is impossible. Figure 6.40 shows the case where A is a vertex and B is an edge. Figure 6.41 shows case where A and B are both edges. To construct one of these bounds, we construct a line which connects an extreme point of A with the extreme point of B on the same side. The line is then rotated about the extreme point of A such that it is interior to B, and its perpendicular distance from the extreme point of B is ϵ_t. The rotated line bounds P.

If A and B do not face each other, then strong reaching is impossible. Weak reaching may be possible, however. For example, Figure 6.42 shows a case where weak reaching is possible between two unconnected parallel edges. Generally, weak reaching positions between edges which do not face each other can be computed as the union of weak reaching positions between vertices of A and the edge B. If the vertices of A do not face B, as is the case in Figure 6.42, then weak reaching positions can be computed as the union of weak reaching positions between vertices of A and vertices of B.

The planar algorithm generalizes fairly easily to three dimensions. We will construct a convex set P that contains the reaching positions of B from A.

For the case where A and B face each other, we must compute normal vectors to both. There are more cases in three dimensions than two. For simplicity, we will

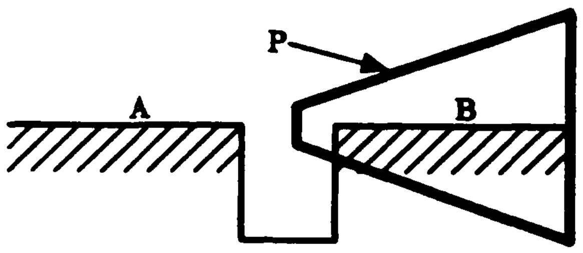


Figure 6.42: P contains the weak reaching positions between the unconnected parallel edges A and B.

not describe them here. We then clip from A the portion that is interior to B, and from B the portion that is interior to A. We continue on, using what remains of the two.

Commanded positions must be behind B. To enforce this constraint, we bound P by a plane which is perpendicular to B's normal vector. For weak reaching, we place this bound in front of B by ϵ_t. For strong reaching, we place this bound behind B by ϵ_p.

For weak reaching, there are two types of lateral bounds. For each bound of the first type, we construct a plane which connects an edge of A with the extreme point of B on the opposite side. This plane is then rotated about the edge of A such that it is exterior to B, and its perpendicular distance from the extreme point of B is ϵ_t. For each bound of the second type, we construct a plane which connects an edge of B with the extreme point of A on the opposite side. This plane is then rotated about the extreme point of A such that it is exterior to B, and its perpendicular distance from the edge of B is ϵ_t.

Strongly reaching a vertex or edge is impossible. When the goal is a face, there are two types of lateral bounds. For each bound of the first type, we construct a plane which connects an edge of A with the extreme point of B on the same side. This plane is then rotated about the edge of A such that it is interior to B, and its perpendicular distance from the extreme point of B is ϵ_t. For each bound of the second type, we construct a plane which connects an edge of B with the extreme point of A on the same side. This plane is then rotated about the extreme point of A such that it is interior to B, and its perpendicular distance from the edge of B is ϵ_t.

Figures 6.43 through 6.45 show examples of reaching computations between unconnected atoms. These examples were all generated by the planner.

If A and B do not face each other, then strong reaching is impossible. Weak reaching may be possible, however. For example, consider Figure 6.42 as a cross section of a three dimensional environment. Here, weak reaching is possible between two unconnected parallel faces. Generally, weak reaching positions between faces which do not face each other can be computed as the union of weak reaching positions between edges of A and the face B. If the edges of A do not face B, as is

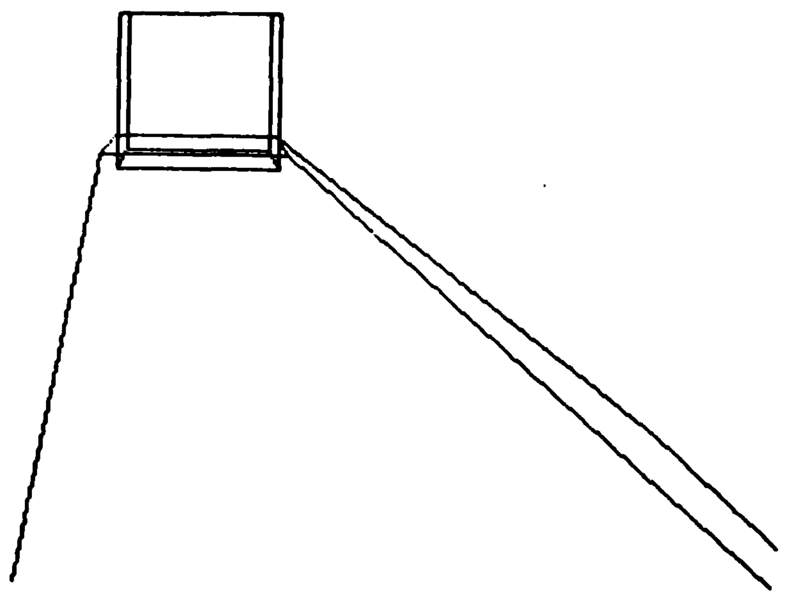


Figure 6.43: The weak reaching positions of the bottom of a cube, when starting from the top left edge, as computed by the planner.

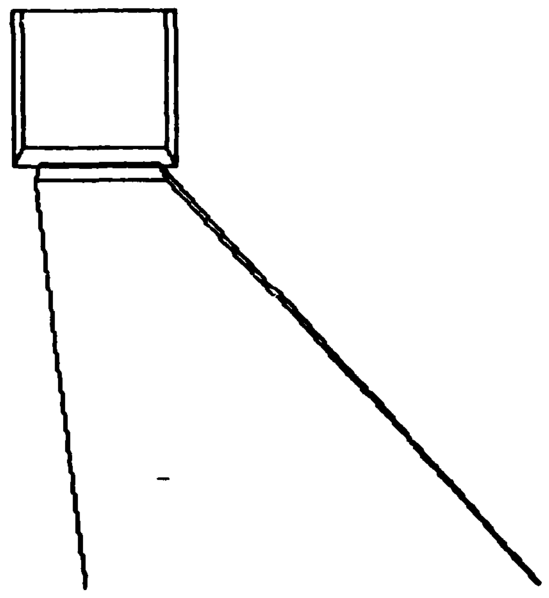


Figure 6.44: The strong reaching positions of the bottom of a cube, when starting from the top left edge, as computed by the planner.

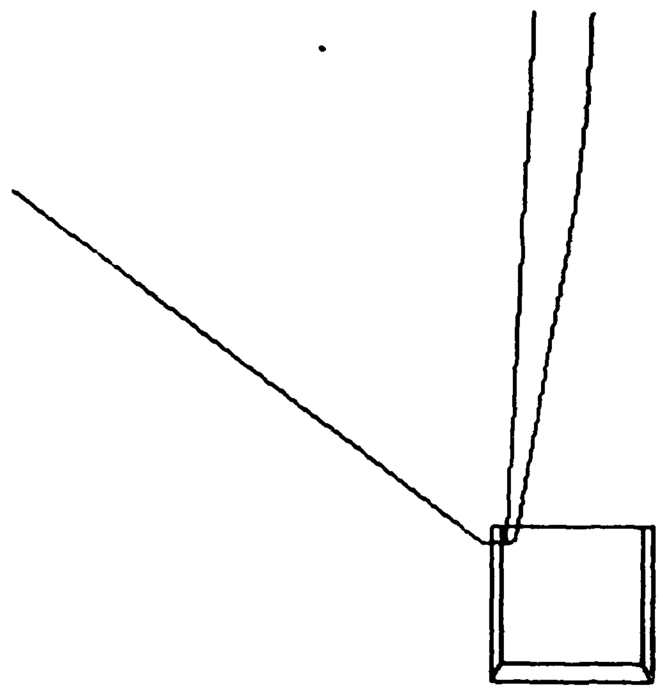


Figure 6.45: The weak reaching positions of the top left edge of a cube, when starting from the bottom, as computed by the planner.

the case in Figure 6.42, then weak reaching positions can be computed as the union of weak reaching positions between edges of A and edges of B.

6.7.6 Sensed Positions

Computing the set of possible sensed positions of an atom A is simply a matter of growing A in three dimensions by ϵ_s. If A is a vertex, we obtain a sphere of radius ϵ_s. If A is an edge, we obtain a cylinder of radius ϵ_s. If A is a face, we obtain a block which is rounded at its vertices. In practice, all of these sets can be approximated by convex polyhedra. Figure 6.46 shows the possible sensed positions of the bottom of a hole.

6.7.7 Sensed Forces

The possible sensed forces of a point is equal to its friction cone, with the cone angle increased by θ_f.

- Figure 2.3 shows the friction cone of a point on face.
- Figure 2.4 shows the friction cone of a point on a convex edge. This cone is the set sum of the friction cones of the containing faces of the edge.

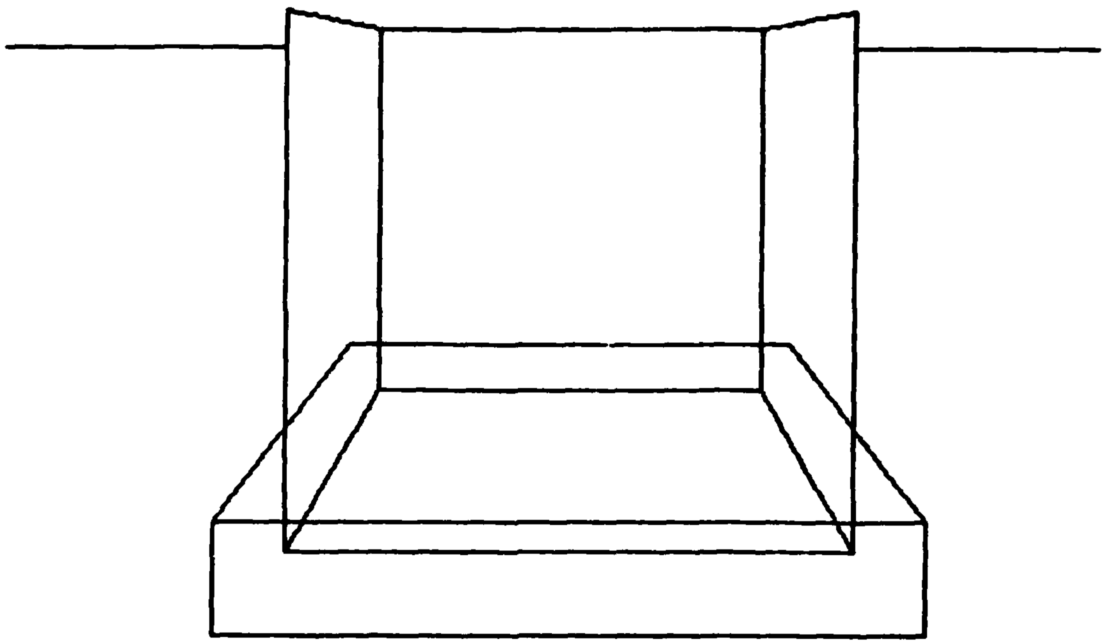


Figure 6.46: The possible sensed positions of the bottom of a hole.

- Figure 2.5 shows the friction cone of a convex vertex. This cone is the set sum of the friction cones of the containing faces of the vertex.

6.8 Computational Complexity

This section examines the worst case time complexity of the planner. The analysis results in an upper bound on the time that it will take for an application. The analysis does not necessarily characterize a realistic example, but it does serve as a warning of what might happen in an arbitrarily chosen example.

We will measure the complexity of a computation by the number of arc operations that it takes. Arc operations include union, intersection, and differencing. An arc is composed of a set of polyhedra. Polyhedral operations can be performed in constant time with respect to the size of the environment.

The central computation of the planner is the expansion of a state. Expanding a state involves computing weak arcs for all component atoms, and composing them together. Thanks to memoization, after the first state has been expanded, most of the atoms in an environment will have been expanded. Thus, expanding subsequent states will take much less time. It is thus reasonable to characterize the complexity of the planner by the complexity of a state expansion.

If we limit the number of atoms per state, then the number of weak arcs per state is equal in order to the number of weak arcs per atom. Thus, it suffices to compute the number of arc operations required to compose all of the weak arcs of an atom.

Let k_a be the number of atoms. The number of weak arcs per atom is equal to k_a times the number of weak arcs from an atom A to an atom B. The maximum number of atoms in a path from A to B is k_a. The maximum number of topologically different paths from A to B is equal to the number of permutations of $k_a - 2$ objects, since the first and last atoms in the path are fixed. Thus, there are at most $(k_a - 2)!$ different paths from A to B. This is a conservative bound, since many paths are impossible, as we shall see below. Using the various types of termination, there are then $O((k_a - 2)!)$ different arcs from A to B. Thus, there are $O(k_a(k_a - 2)!)$ weak arcs per state.

6.8.1 Sticking Termination

Type 1 arcs have a useful property: If two type 1 arcs have identical targets, then they can be unioned together into a single arc. This property allows us to combine the weak type 1 arcs of a state into only $O(k_a)$ different arcs. $O(k_a(k_a - 2)!)$ arc operations are required to do this for the first state.

Composing k_a different weak type 1 arcs results in $O(2^{k_a})$ different arcs. The target of each resultant arc is a subset of the k_a weak targets. The composition can be performed in $O(2^{k_a-1})$ arc operations, using the algorithm of Section 6.6.2. The algorithm inserts each arc into a nonoverlapping queue. The first arc is inserted into the queue in 0 arc operations, resulting in a queue of length 1. The second arc is inserted into the queue in $O(1)$ arc operations, resulting in a queue of maximal length 3. The third arc is inserted into the queue in $O(3)$ arc operations, resulting in a queue of maximal length 7. The k_ath arc is inserted into the queue in $O(2^{k_a-1})$ arc operations, resulting in a queue of length $O(2^{k_a})$.

The total number of arc operations required to expand the first state is

$$O(k_a(k_a - 2)! + 2^{k_a-1}). \tag{6.3}$$

Although this expression is exponential, in practice the number of arc operations is expected to be much less. There are two exponential terms in the expression. The first, $O((k_a - 2)!)$, represents the number of topologically different paths from an atom A to an atom B. Many of these paths can be pruned by the following constraints:

- An edge or vertex can only follow an atom that it is physically connected to.
- Faces which are occluded from each other cannot follow each other in a path.

The second exponential term, $O(2^{k_a-1})$, represents the number of different target sets that the robot can terminate in. Most target sets are not feasible, under the constraint that they must be reachable under a common commanded position.

6.8.2 Sensing Termination

In general, a pair of type 2 or 3 arcs which have identical targets cannot be unioned together. We will show this for type 2 arcs. Suppose that $\{\alpha_1, \alpha_2, \alpha_3\}$ is a reliable set of type 2 arcs. α_1 and α_2 have identical targets; α_3 has a different target than the other two. The cpositions of α_1 and α_3 intersect, but the spositions do not. The spositions of α_2 and α_3 intersect, but the cpositions do not. Since the unioned arc $\alpha_1 \bigcup \alpha_2$ intersects α_3, the set $\{\alpha_1 \bigcup \alpha_2, \alpha_3\}$ is unreliable.

Thus, we are stuck with $O(k_a(k_a - 2)!)$ weak arcs per state. Composing these arcs results in $O(2^{k_a(k_a-2)!})$ different arcs. Each resulting arc is a subset of the $k_a(k_a - 2)!$ weak arcs. The composition can be performed in

$$O(2^{k_a(k_a-2)!-1}) \tag{6.4}$$

arc operations. This expression also represents the total number of arc operations required to expand the first state using type 2 or 3 termination.

6.8.3 Conclusion

In the worst case, expanding a state under sticking termination may require an exponential number of arc operations. It is expected that in practice, the actual number of arc operations required is much less.

In the worst case, expanding a state under sensing termination may require a doubly exponential number of arc operations. This gives us reason to believe that in practice this computation is much more difficult than sticking termination. Our experiments supported this belief.

6.9 Summary

In this chapter, we described a compliant motion planner. The input to the planner is a polyhedral environment representing the configuration space of a robot which can translate in three dimensions, along with surface start and goal regions. The planner attempts to compute an executable graph called a state graph which contains generalized spring motions with conditional tests after each motion. The motions may be terminated by either sticking or sensing.

The planner characterizes the state of the robot as a set of atoms where the robot might be located under bounded sensing and control uncertainty. An atom is a set of configurations. The set of atoms is a partition of configuration space into complete, nonoverlapping subsets. We chose an atom decomposition in which atoms are vertices, edges, and faces from the environment, plus one atom for all of free space.

The planner operates by expanding the arcs of heuristically chosen states, and testing the resulting state graph using an AND/OR tree evaluator. To expand a state, the planner computes all possible motions from the state, grouping them into weak arcs, which have a common target atom. Weak arcs are then composed into reliable strong arcs, which share a common set of target atoms. A strong arc points at states that are composed of confusable collections of its target atoms. On execution, the robot decides which target state it has reached using position and force sensing.

The planner was tested on three configuration space environments: a cube, a single hole, and a double hole. For each test, the planner produced a motion strategy suitable for execution on a robot. The single hole strategy was successfully executed on an IBM 7565 robot.

Chapter 7

Future Work

This chapter presents ideas for future work.

7.1 Automatic Atom Decomposition

A critical component of the planner input is a decomposition of the environment into faces. This decomposition is used to generate atoms. The success of the planner depends on the atom decomposition. We need an algorithm for performing the atom decomposition automatically.

Figure 6.13 showed an example where an atom was so large that a reliable strategy was overlooked. Referring to the figure, one way to solve this task is to make the region $B \subset A$ an atom. This reduces the positional uncertainty of the robot such that the obstacles O_1 and O_2 can be avoided by the forward projection of the start region under the commanded position $\mathbf{p}$.

7.1.1 Subset Atoms

In thinking about atom decomposition in tasks like that of Figure 6.13, one wonders to what extent a robot can attain an accurate placement B on a face A once it has reached an arbitrary point of A. It turns out that under certain conditions, the robot can position itself on a face to within ϵ_t of a desired position $\mathbf{x}_d$, starting at any point of A. To do this, we command a generalized spring position $\mathbf{p}$ which causes the robot to slide across the face within a cylinder of radius ϵ_t toward $\mathbf{x}_d$. The sliding trajectory is terminated by sensing a position that is within ϵ_t of $\mathbf{x}_d$. This technique works under the following conditions:

1. The commanded position $\mathbf{p}$ must be past $\mathbf{x}_d$, such that the weak sticking region of A due to $\mathbf{p}$ does not intersect the desired sliding trajectory. Otherwise, premature termination due to sticking may occur. Whether this is possible depends on the friction cone of A. For example, if A has infinite friction, then sliding is impossible.

2. $\mathbf{x}_d$ must be within the border of A by at least ϵ_t, so that the robot does not fall off of A while sliding to $\mathbf{x}_d$.

Our conclusion is that on most faces we can arbitrarily define *subset atoms*, circles of radius ϵ_t that are reachable from every point on the face. The idea is that a motion strategy which reaches the face should proceed to a subset atom before exiting the face. This reduces the size of forward projections from the face. Choosing the placement of subset atoms is an open problem.

7.1.2 Dynamic Decomposition

A *dynamic decomposition* would compute the atom decomposition while planning. For example, the planner might be able to use forward or backward projections to determine the placement of subset atoms.

7.1.3 Static Decomposition

A *static decomposition* would decompose an environment into faces prior to planning. For example, consider the hole problem of Figure 5.2, and its solution in Figures 6.10 and 6.11. The surface above the hole has been decomposed into four faces. The start region is an edge on a face which is aligned with the hole, and of the same width. If the width of that face were any larger, the solution of the problem would have been much more difficult. It is difficult to slide into the hole from one of the two large faces above the hole. The reason is that a motion which attempts to slide into the hole may also slide onto one of the two smaller faces above the hole. A possible solution to this problem is a decomposition in which the interior angles at vertices are always acute. Figure 7.1 shows a new decomposition of the hole environment using this approach. This approach seems to work for sliding trajectories, but does not help the free space problem of Figure 6.13.

7.2 Modeling Error

We have allowed for errors in the sensing and control of the robot, but have ignored geometric errors in the parts of an assembly. These errors could be due to either physical variations in parts, or geometric modeling errors.

One approach would be to treat these errors as parametric errors, and establish bounds on them. For example, polyhedral models can be seen as data structures of vertices. We could model *vertex uncertainty*, in which uncertainty balls are used to bound the possible locations of vertices in configuration space. These uncertainty balls would define a range of orientations for each face. To compute sticking and sliding on a face, we would need to include uncertainty in the orientation of the face in the weak and strong sticking cones. The uncertainty balls would also define a range of volumes for each obstacle. For example, Figure 7.2 shows the maximal volume for an obstacle. To compute collision free trajectories, we could represent all obstacles by their maximal volume. Alternatively, to compute all reachable contact points on an obstacle B, we could represent all obstacles $O \neq B$ by their minimal

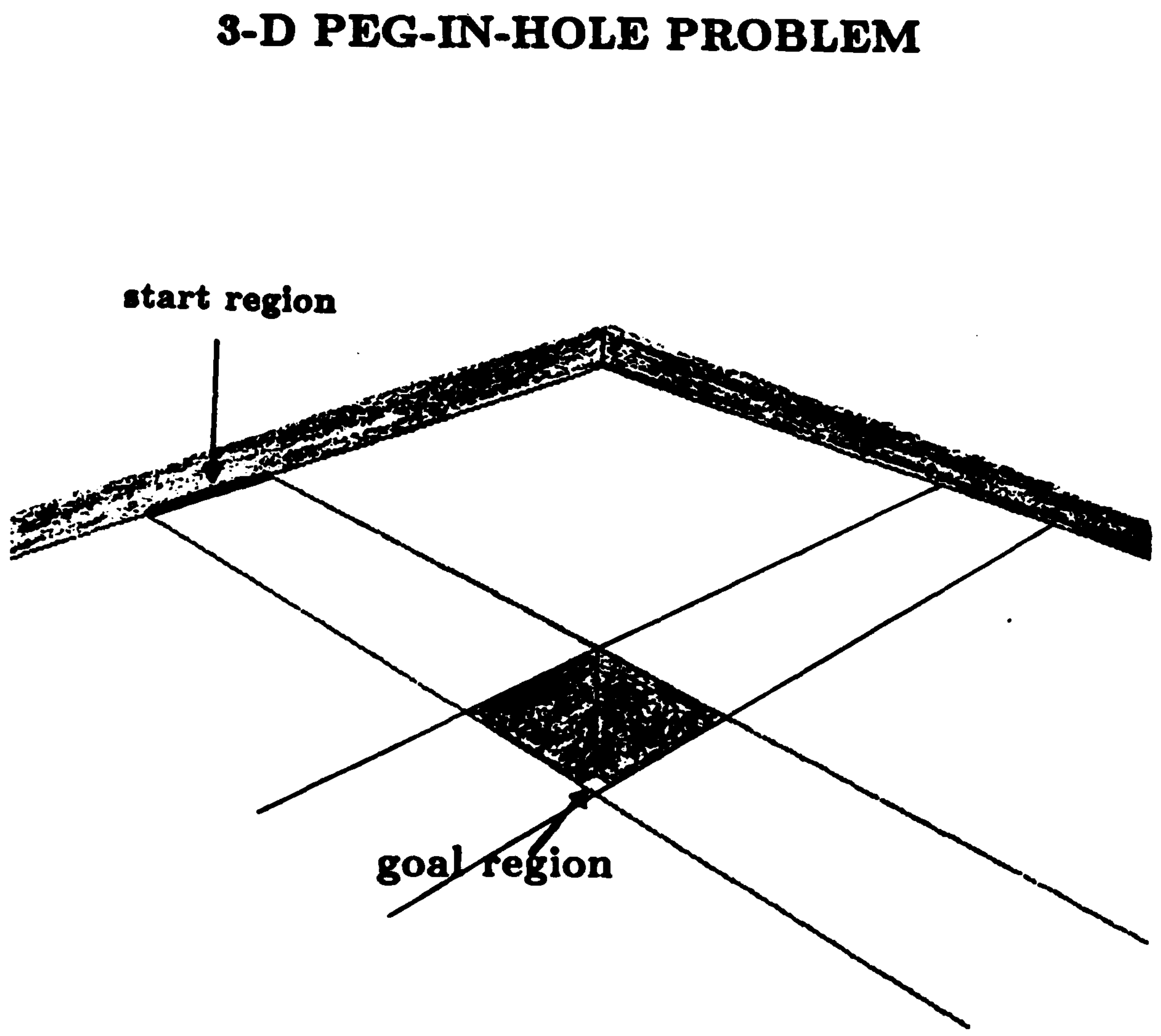


Figure 7.1: An environment consisting of a square hole in an obstacle surface. All interior face angles are acute.

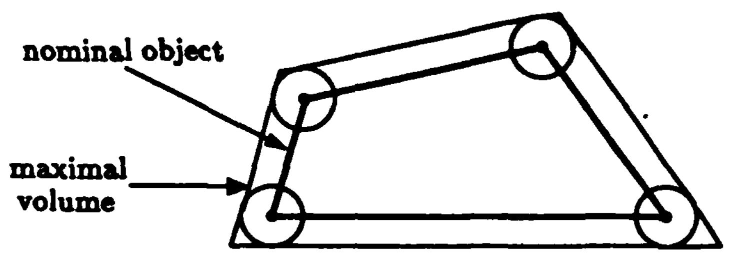


Figure 7.2: An object and its maximal volume under vertex uncertainty.

volume. This would maximize free space travel to B. Many details would have to be considered to complete this approach.

Another approach, currently being pursued by Donald [1986], is to represent variations in the parts by additional configuration space dimensions. Many configuration space planning concepts, such as pre-images, extend naturally to the new space. Some problems caused by higher dimensional configuration spaces are described in Section 7.5.

7.3 Free Space Goals

The programming system allows motions whose start and goal regions are in contact space. It would be nice to have the option of planning motions which start and stop in free space.

This would not be difficult to implement for the teaching system. The teaching system computes backprojections by constructing reachable volumes in configuration space, and performing hidden surface analysis to remove unreachable contact points. Instead of returning contact regions, we could return volumes of free space between contact regions and the backprojection base.

Extending the planner to free space goals would be more difficult. Currently, we group all of free space into one atom. We would have to decompose free space into multiple atoms. This decomposition could be done in many ways. This task falls into the realm of automatic atom decomposition (Section 7.1). The collision-free motion planning literature has been considering this decomposition problem for some time. See Yap [1985] for a survey.

7.4 Rotations

Since many applications require rotations, extensions are needed in this area. Rotations create a number of problems for our algorithms. The first problem is that rotations add more dimensions to the configuration space. Adding a complete set of rotations to a three-dimensional robot results in a six-dimensional configuration space. Aspects of this problem are examined in Section 7.5.

7.4.1 Planar Cartesian Robots with Rotation

If we examine the case of a planar Cartesian robot which can rotate, then we can consider the effects of rotation in a three-dimensional configuration space. A configuration space of this type consists of two translational dimensions and one rotational dimension.

Euclidean Approximation

Rotation spaces wrap around. If we rotate an object incrementally in the positive or negative direction, we will eventually get back to where we started. Since our system assumes Euclidean space coordinates, we could approximate one-dimensional rotations by a subset of $\Re^1$. In order to avoid redundancy, we could choose the subset $[-\pi, \pi]$. Unfortunately, this would eliminate trajectories which achieve a desired orientation by rotating beyond $-\pi$ or π. However, many tasks could be performed within this limitation.

When constructing a three-dimensional space where one of the dimensions represents rotations, we are faced with the question of how to scale rotational coordinates so that they relate meaningfully to translational coordinates. Erdmann [1984] observed that the normal vector at a contact point should represent the reaction force of the surface in the absence of friction. He showed that this property can be satisfied as follows:

1. The reference point of the robot should be chosen at its center of mass. This ensures that pure forces at the reference point result in pure translations, and that pure torques result in pure rotations.
2. Rotational coordinates should be scaled by ρ, the *radius of gyration* of the robot, i.e., the distance from the reference point of the robot at which we would place its total mass in order to obtain the correct moment of inertia.

A configuration in this configuration space is thus given by the coordinates (x, y, q), where $q = \rho\theta$. A rotational force is given by τ/ρ, where τ is the corresponding real space torque.

Position Sensing Errors

The maximum position sensing error in this configuration space is equal to

$$\epsilon_s = \sqrt{\epsilon_{s,t}^2 + (\rho\epsilon_{s,r})^2}, \tag{7.1}$$

where $\epsilon_{s,t}$ is the maximum real space position sensing error, and $\epsilon_{s,r}$ is the maximum real space rotational sensing error.

Force Sensing Errors

The maximum angular force sensing error θ_f in this configuration space is equal to $\arctan(\epsilon_f)$, where ϵ_f is the maximum distance between the tip of a unit force vector and the tip of a resultant sensed force vector in configuration space. ϵ_f is given by

$$\epsilon_f = \sqrt{\epsilon_{f,t}^2 + (\epsilon_{f,r}/\rho)^2}, \tag{7.2}$$

where $\epsilon_{f,t}$ is the maximum sensing error in a unit force in real space, and $\epsilon_{f,r}$ is the maximum sensing error in a unit torque in real space.

Positioning Errors

The maximum positioning error ϵ_p in this configuration space is equal to

$$\sqrt{\epsilon_{p,t}^2 + \epsilon_{p,r}^2}, \tag{7.3}$$

where $\epsilon_{p,t}$ is the maximum real space positioning error, and $\epsilon_{p,r}$ is the maximum real space rotational error.

Velocity Errors

The maximum angular velocity error θ_v in this configuration space is equal to $\arctan(\epsilon_v)$, where ϵ_v is the maximum distance between the tip of a commanded unit velocity vector and a resultant velocity vector in free space. ϵ_v is given by

$$\epsilon_v = \sqrt{\epsilon_{v,t}^2 + \epsilon_{v,r}^2}, \tag{7.4}$$

where $\epsilon_{v,t}$ is the maximum error in a commanded unit translational velocity in real space, and $\epsilon_{v,r}$ is the maximum error in a commanded unit rotational velocity in real space.

Friction

Erdmann showed that friction cones in this three-dimensional configuration space are actually two-dimensional regions. This is because rotations are not subject to friction. Our algorithms could be modified to work with these friction cones instead of those that are presently used.

Curved Faces

When rotations are allowed, configuration space faces are curved. The reason is that as we rotate the robot about a point contact, the reference point of the robot traces out a circular arc in real space. The configuration space face is composed of a collection of these circular arcs, one for each of point contacts which generate the configuration space face.

We rely on the flatness of faces in our algorithms. Forward projections in free space use visibility computations which depend on flat faces. Forward projections in contact space assume that friction cones are uniform across a face. This problem could be solved by decomposing curved faces into collections of flat faces, and using geometric modeling error (Section 7.2) to account for the approximation error.

7.4.2 Pure Three-Dimensional Rotations

If we examine the case of a three-dimensional Cartesian robot which can only rotate, then we can consider the effects of pure rotations in a three-dimensional configuration space. Rotations can be expressed as three-dimensional vectors from the special orthogonal group $SO(3)$. Canny [1986] showed that straight lines in this space correspond to approximately uniform rotations of a Cartesian robot. Erdmann [1984] showed that coordinates in this space should be scaled respectively by ρ_1, ρ_2, and ρ_3, the radii of gyration about the three orthogonal principle axes of the robot. Again, this ensures that surface normals represent reaction forces in the absence of friction.

If we denote the coordinates of a rotation by $(\theta_1, \theta_2, \theta_3)$, then the coordinates of the corresponding configuration are $(\rho_1\theta_1, \rho_2\theta_2, \rho_3\theta_3)$. Maximal errors can be computed as we did in Section 7.4.1. Faces are curved, and would have to be decomposed into flat faces with geometric modeling error.

There are no friction cones in this space, although there are reaction forces due to surface normals. Note that at edges and vertices, surface normals are sets given by summing the possible reaction forces of the containing faces. Thus, even without friction, it is possible to stick on a surface.

7.5 Higher Dimensional Configuration Spaces

Many extensions to the programming system would add new dimensions to the configuration space. These extensions include:

- Rotations.
- Grasping and releasing. A parallel jaw gripper adds one dimension. A multi-fingered hand such as the Salisbury hand [Salisbury 1982] or the Utah-MIT hand [Jacobsen 1984] adds several more dimensions.
- Geometric modeling error.

7.5.1 Slices

One approach would be to constrain motion to only three dimensions at a time. For example, an application might be implemented as a sequence of independent translations and rotations.

To plan translations, we would decompose the six-dimensional configuration space into three-dimensional *slices* [Lozano-Pérez 1983a]. Each slice would represent three-dimensional translations over a particular range of orientations. Geometric modeling error (Section 7.2) could be used to account for the variation within a slice. The method of Chapter 6 could be used to plan motions within each slice in this space.

To plan rotations, we would decompose the configuration space into three-dimensional slices representing three-dimensional rotations over a particular range of translations. The method introduced in Section 7.4.2 could be used to plan motions within each slice in this space.

There is reason to think that pure rotations about the reference point of the robot are limited without allowing free space goals (Section 7.3). Requiring termination in contact configurations would seem to severely limit the types of reorientation that can occur in a pure rotation.

7.5.2 Direct Computation

A second approach would be to work directly in the higher dimensional configuration space. This approach would require major reformulations of the algorithms described in this thesis.

7.6 Optimal Strategies

The planner returns the first reliable strategy that it can find. For our implementation, finding the best strategy would have taken too much time. The next section describes an approach to speed up the planner. If the planner were fast enough, we might want to consider how we would find optimal strategies. First, we would have to devise an evaluation function for motion strategies. Then, we would have to develop an algorithm for finding the best motion strategy under the given evaluation function.

7.7 Parallel Implementation

Currently, the planner takes quite a bit of time. To obtain experimental results, we had to limit the number of atoms per state (Section 6.4). One approach to speed up the planner would be to parallelize it. To do this, we would have to identify modules within the planner which could run in parallel. Here are some initial thoughts:

- One of the tasks of the planner is to compute all possible motions from each atom. We could process each atom in parallel. Since these calculations depend on each other, an atom would sometimes have to wait for others to complete. However, many calculations would be independent of other atoms, such as partition composition, and could be performed in parallel. Note that a method for avoiding deadlock would be required.

- Once all atoms have been processed, the arcs of a state can be expanded independently. We could assign one processor to each state.

References

An, C., "Trajectory and Force Control of a Direct Drive Arm", Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, also AI-TR-912, MIT Artificial Intelligence Laboratory, September, 1986.

Andreae, P., "Justified Generalization: Acquiring Procedures From Examples", Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, also AI-TR-834, MIT Artificial Intelligence Laboratory, January, 1985.

Arnold, V., *Mathematical Models of Classical Mechanics*, Springer-Verlag, New York, 1978.

Asada, H., "Studies on Prehension and Handling by Robot Hands with Elastic Fingers", Ph.D. dissertation, University of Kyoto, 1979.

Baumeister, T., editor, *Marks' Standard Handbook for Mechanical Engineers*, McGraw-Hill, 1978.

Bonner, S., and K. Shin, "A Comparative Study of Robot Languages", *IEEE Computer*, December, 1982.

Brooks, R., "Symbolic Error Analysis and Robot Planning", *International Journal of Robotics Research*, Vol. 1, No. 4, 1982.

Brooks, R., "Solving the Find-Path Problem by Good Representation of Free Space", *IEEE Transactions on Systems, Man, and Cyberbetics*, Vol. 13, 1983.

Brooks, R., and T Lozano-Pérez, "A Subdivision Algorithm in Configuration Space for Findpath with Rotation", *Eighth International Joint Conference on Artificial Intelligence*, Karlsruhe, Germany, August, 1983.

Brost, R., "Automatic Grasp Planning in the Presence of Uncertainty", *IEEE International Conference on Robotics and Automation*, San Francisco, April, 1986.

Buckley, S., "A Structured Programming Robot Language", with G. Collins, in S. Nof, *Handbook of Industrial Robotics*, John Wiley and Sons, 1985, pp. 381-403.

Caine, M., "Chamferless Assembly of Rectangular Parts in Two and Three Dimensions", S.M. dissertation, MIT Department of Machanical Engineering, June 1985.

Canny, J., "Collision Detection for Moving Polyhedra", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 8, No. 2, March, 1986.

Canny, J., and J. Reif, "New Lower Bound Techniques for Robot Motion Planning Problems", to be published, 1986.

Cutkosky, M., "Grasping and Fine Manipulation for Automated Manufacturing", Ph.D. dissertation, Carnegie-Mellon University, January, 1985.

Donald, B., "Local and Global Techniques for Motion Planning", S.M. dissertation, MIT Department of Electrical Engineering and Computer Science, also AI-TR-791, MIT Artificial Intelligence Laboratory, 1984. A subset is to appear in *Artificial Intelligence*.

Donald, B., "On Motion Planning with Six Degrees of Freedom: Solving the Intersection Problems in Configuration Space", *IEEE International Conference on Robotics and Automation*, St. Louis, March, 1985.

Donald, B., "Robot Motion Planning with Uncertainty in the Geometric Models of the Robot and Environment: A Formal Framework for Error Detection and Recovery", *IEEE International Conference on Robotics and Automation*, San Francisco, April, 1986.

Draper Laboratories, Fourth Annual Seminar on Robotics and Advanced Assembly Systems, Cambridge, Massachusetts, November, 1983.

Dufay, B., and J. Latombe, "An Approach to Automatic Robot Programming Based on Inductive Learning", in Brady, M., and R. Paul, *Robotics Research: The First International Symposium*, MIT Press, 1984.

Erdmann, M., "On Motion Planning With Uncertainty", S.M. dissertation, MIT Department of Electrical Enginering and Computer Science, also AI-TR-810, MIT Artificial Intelligence Laboratory, 1984.

Erdmann, M., "Using Backprojections for Fine Motion Planning with Uncertainty", *International Journal of Robotics Research*, Vol. 5, No. 1, Spring, 1986.

Erdmann, M., and M. Mason, "An Exploration of Sensorless Manipulation", *IEEE International Conference on Robotics and Automation*, San Francisco, April, 1986.

Fearing, R., "Simplified Grasping and Manipulation With Dextrous Hands", *American Control Conference*, San Diego, California, June, 1984.

Finkel, R., R. Taylor, R. Bolles, and J. Feldman, "AL: A Programming System for Automation", AIM-243, Stanford Artificial Intelligence Laboratory, Stanford University, November, 1974.

Grossman, D., and R. Taylor, "Interactive Generation of Object Models with a Manipulator", *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 8, No. 9, September, 1978.

Grossman, D., "Programming a Computer Controlled Manipulator by Guiding It Through the Motions", IBM Thomas J. Watson Research Center, IBM Research Report RC-6393, March, 1977.

Hirzinger, G., and K. Landzettel, "Sensory Feedback Structures for Robots with Supervised Learning", *IEEE International Conference on Robotics and Automation*, St. Louis, March, 1985.

Hogan, N., "Impedance Control of Industrial Robots", *Robotics and Computer-Integrated Manufacturing*, Vol. 1, No. 1, 1984.

Inoue, H., "Force Feedback in Precise Assembly Tasks", MIT Artificial Intelligence Laboratory, AIM-308, August, 1974.

Jacobsen, S., J. Wood, D. Knutti, and K. Biggers, "The Utah/MIT Dextrous Hand: Work in Progress", in Brady, M., and R. Paul, *Robotics Research: The First International Symposium*, MIT Press, 1984.

Kerr, J., "Issues Related to Grasping by Robots", Ph.D. dissertation, Department of Mechanical Engineering, Stanford University, January, 1985.

Koutsou, A., "A Geometric Reasoning System for Moving an Object While Maintaining Contact with Others", *ACM Symposium on Computational Geometry*, Yorktown Heights, N.Y., 1985.

Laugier, C., "A Program for Automatic Grasping of Objects with a Robot Arm", *Eleventh Symposium of Industrial Robots*, Japan Society of Biomechanisms and Japan Industrial Robot Association, 1981.

Laugier, C., and P. Theveneau, "Planning Sensor-Based Motions For Part-Mating Using Geometric Reasoning Techniques", *European Conference on Artificial Intelligence*, Brighton, July, 1986.

Lieberman, L., and M. Wesley, "AUTOPASS: An Automatic Programming System for Computer Controlled Mechanical Assembly", *IBM Journal of Research Development*, Vol. 21, No. 4, 1977, pp. 321-333.

Lozano-Pérez, T., "The Design of a Mechanical Assembly System", S.M. dissertation, MIT Department of Electrical Engineering and Computer Science, also AI-TR-397, MIT Artificial Intelligence Laboratory, 1976.

Lozano-Pérez, T., and M. Wesley, "An Algorithm for Planning Collision Free Paths Among Polyhedral Obstacles", *Communications of the ACM*, October, 1979.

Lozano-Pérez, T., "Automatic Planning of Manipulator Transfer Movements", *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 11, No. 10, 1981.

Lozano-Pérez, T., "Spatial Planning: A Configuration Space Approach", *IEEE Transactions on Computers*, February, 1983a.

Lozano-Pérez, T., "Robot Programming", *IEEE Proceedings*, 1983b.

Lozano-Pérez, T., M. Mason, and R. Taylor, "Automatic Synthesis of Fine-Motion Strategies for Robots", *International Journal of Robotics Research*, Vol. 3, No. 1, Spring, 1984.

Lozano-Pérez, T., "Motion Planning For Simple Robot Manipulators", *Third International Symposium on Robotics Research*, Paris, October, 1985.

Luenberger, D., *Introduction to Dynamic Systems*, John Wiley and Sons. 1979.

Mani, M., and W. Wilson, "A Programmable Orienting System for Flat Parts", *Proceedings NAMRII XII*, 1985.

Mason, M., "Compliance and-Force Control for Computer Controlled Manipulators", AI-TR-515, MIT Artificial Intelligence Laboratory, April, 1979.

Mason, M., "Manipulator Grasping and Pushing Operations", Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, also AI-TR-690, MIT Artificial Intelligence Laboratory, 1982.

Mason, M., "Automatic Planning of Fine Motions: Correctness and Completeness", *IEEE International Conference on Robotics*, Atlanta, 1984.

Mason, M., "Mechanics and Planning of Manipulator Pushing Operations", *International Journal of Robotics Research*, Vol. 5, Number 3, Spring, 1986.

McLaughlin, J., "TRIG: An Interactive Robotic Teach System", MIT AI Working Paper 234, June, 1982.

Nguyen, V., "The Synthesis of Stable Grasps in the Plane", *IEEE International Conference on Robotics and Automation*, San Francisco, April, 1986.

Ohwovoriole, M., and B. Roth, "A Theory of Parts Mating For Assembly Automation", *Proceedings of the Robot and Man Symposium 81*, Warsaw, Poland, September 1981.

Paul, R., *Robot Manipulators*, MIT Press, Cambridge, Massachusetts, 1981.

Peshkin, M., "Planning Robotic Manipulation Strategies for Sliding Objects", Ph.D dissertation, Department of Physics, Carnegie-Mellon University, 1986.

Peshkin, M., and A. Sanderson, "Reachable Grasps on a Polygon: The Convex Rope Algorithm", *IEEE Journal of Robotics and Automation*, Volume 2, Number 1, March, 1986.

Raibert, M., and J. Craig, "Hybrid Position/Force Control of Manipulators", *Journal of Dynamic Systems, Measurement, and Control*, No. 102, June, 1981, pp. 126-133.

Salisbury, J.K., "Active Stiffness Control of a Manipulator in Cartesian Coordinates", *IEEE Conference on Decision and Control*, Albuquerque, New Mexico, November, 1980.

Salisbury, J.K., "Kinematic and Force Analysis of Articulated Hands", Ph.D. dissertation, Stanford University, Department of Mechanical Engineering, 1982.

Schumacker, R., B. Brand, M. Gilliland, and W. Sharp, "Study for Applying Computer-Generated Images to Visual Simulation", AFHRL-TR-69-14, U.S. Air Force Human Resources Laboratory, September 1969.

Schwartz, J., and M. Sharir, "On the Piano Movers Problem, II: General Techniques for Computing Topological Properties of Real Algebraic Manifolds", Courant Institute of Mathematical Sciences, Report No. 41, 1982.

Sedgewick, R., *Algorithms*, Addison-Wesley, 1983.

Segre, Alberto Maria, and G. DeJong, "Explanation-Based Manipulator Learning: Acquisition of Planning Ability Through Observation", *IEEE International Conference on Robotics and Automation*, St. Louis, March, 1985.

Selfridge, M., and A. Levas, "Teaching Robots by Example", *Proceedings 13th International Symposium on Industrial Robots*, Chicago, Illinois, April, 1983, pp. (13-160)-(13-165).

Shamos, M., "Geometric Complexity", *Proc. 7th ACM Annual Symposium on the Theory of Computing*, May 1975, pp. 224-233.

Simunovic, "An Information Approach to Parts Mating", Ph.D. dissertation, Department of Electrical Engineering, Massachusetts Institute of Technology, 1979.

Summers, P., and D. Grossman, "XPROBE: An Experimental System for Programming Robots by Example", *International Journal of Robotics Research*, Vol. 3, No. 1, 1984.

Sutherland, I., and G. Hodgman, "Reentrant Polygon Clipping", *Communications of the ACM*, Vol. 17, No. 1, January 1974.

Sutherland, I., R. Sproull, and R. Schumacker, "A Characterization of Ten Hidden-Surface Algorithms", *ACM Computing Surveys*, Vol. 6, No. 1, March 1974.

Taylor, R., "A Synthesis of Manipulator Control Programs from Task-Level Specifications", Ph.D. dissertation, Stanford University, also AIM-282, Stanford Artificial Intelligence Laboratory, 1976.

Taylor, R., P. Summers, and J. Meyer, "AML: A Manufacturing Language", *International Journal of Robotics Research*, Vol. 1, No. 3, 1982.

Taylor, R., "An Integrated Robot System Architecture", Research Report RC 9824 (42454), IBM T. J. Watson Research Center, January, 1983.

Turk, M., "A Fine-Motion Planning Algorithm", *SPIE Conference on Intelligent Robots and Computer Vision*, Cambridge, Massachusetts, September, 1985.

Udupa, S., "Collision Detection and Avoidance in Computer Controlled Manipulators", Ph.D. dissertation, Department of Electrical Engineering, California Institute of Technology, 1977.

Unimation Inc, "User's Guide to VAL: A Robot Programming and Control System", Danbury, Conn., version 12, June 1980.

Vaaler, E., and W. Seering, "Design of a Cartesian Robot", *Robotics and Manufacturing Automation Symposium*, ASME Winter Annual Meeting, Miami Beach, Florida, November, 1985.

Valade, J., "Automatic Generation of Trajectories for Assembly Tasks", *Sixth European Conference on Artificial Intelligence*, Pisa, Italy, September, 1984.

Valade, J., "Raisonnement Geometrique et Synthese de Trajectoire D'assemblage", Doctor of Engineering dissertation, Paul Sabatier University, Toulouse, France, 1985.

Weiler, K., and P. Atherton, "Hidden Surface Removal Using Polygon Area Sorting", *Computer Graphics*, Vol. 11, No. 2, 1977.

Whitney, D., "Resolved Motion Rate Control of Manipulators and Human Prostheses", *IEEE Transactions on Man-Machine Systems*, Vol. 10, pp. 47-53, 1969.

Whitney, D., "Force Feedback Control of Manipulator Fine Motions", *Journal of Dynamic Systems, Measurement, and Control*, June, 1977, pp. 91-97.

Whitney, D., "Quasi-Static Assembly of Compliantly Supported Rigid Parts", *Journal of Dynamic Systems, Measurement, and Control*, Vol. 104, March 1982.

Whitney, D., "Historical Perspective and State of the Art in Robot Force Control", *IEEE International Conference on Robotics and Automation*, St. Louis, March, 1985.

Will, P., and D. Grossman, "An Experimental System For Computer-Controlled Mechanical Assembly", *IEEE Transactions on Computers*, Vol. 24, No. 9, 1975.

Will, P., personal communication regarding industrial studies, 1981.

Winston, P., *Artificial Intelligence*, Addison-Wesley, 1984.

Yap, C., "Algorithmic Motion Planning", to appear in *Advances in Robotics: Volume 1*, edited by J. Schwartz and C. Yap, Lawrence Erlbaum Associates, 1985.

Appendix A

Visibility Analysis

The primary application of visibility analysis is computer graphics. Computer graphics involves computing and displaying the surface regions of a three dimensional environment that are visible to a viewer looking along a viewing axis. The surface regions that the viewer can see depend upon the imaging system used. In this appendix, we develop an algorithm which solves this problem for several different types of imaging systems. Our algorithm is neither new nor optimal. It is provided to establish background vocabulary and nominal algorithms for Chapters 4 and 5, and as a tutorial for readers unfamiliar with the topic.

A.1 Orthographic Imaging Systems

Consider the *orthographic imaging system* depicted in Figure A.1. The viewer looks at an environment of polyhedral objects through a flat, circular lens, of radius 1. Since the lens is flat, the viewer can only see surface regions in the cylinder of the lens. An *image plane* with an associated xy coordinate system is placed in the plane of the lens, as shown in the figure. The negative z-axis is extended along

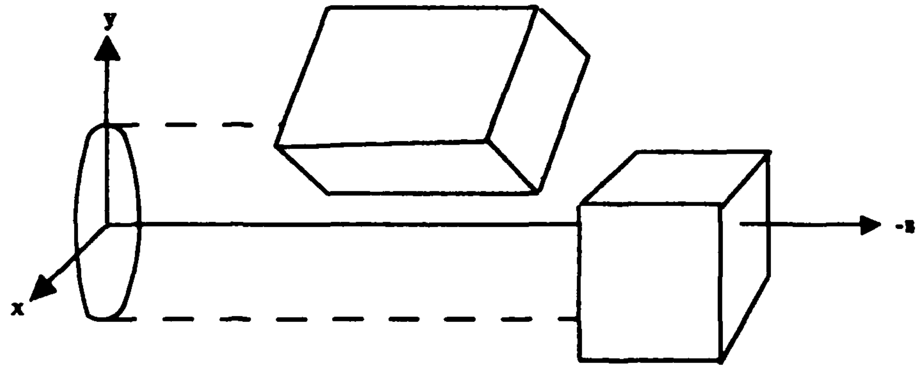


Figure A.1: An orthographic imaging system. The viewer looks at an environment of polyhedral objects through a flat, circular lens.

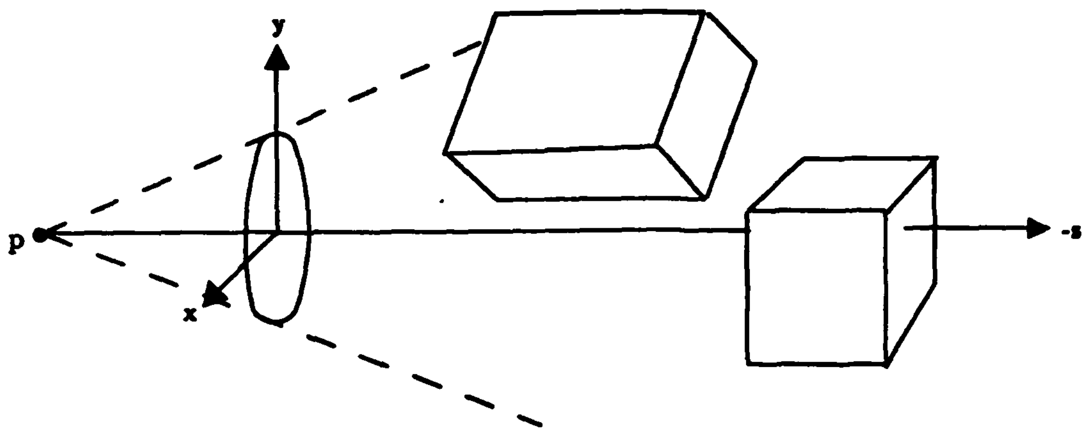


Figure A.2: A convex imaging system. The viewer is located at the focal point **p**.

the viewing axis, completing a right handed coordinate system. Using this set of coordinate axes, the environment can be described in *eye coordinates* by applying a standard homogeneous transformation to its vertices.

Visible surface regions can be computed in this imaging system as follows. Surface regions that are behind the viewer and outside of the cylinder are eliminated by a process called *polygon clipping*, which is described in Section A.6. All vertices are then projected onto the image plane by *orthographic projection*, which in this case means simply ignoring their z-coordinates. Surface regions that are occluded by closer surface regions are clipped away. This process is called *hidden surface removal*, and is described in Section A.5. The surface regions that remain are visible.

Orthographic projections of visible surface regions are sufficient to display them on a video screen. If the actual three dimensional visible surface regions are needed, then the orthographic projection must be inverted. This can be done by keeping the z-coordinates of projected vertices around, and computing backprojected z-coordinates for any new vertices that are created by intersections in the image plane during clipping.

The process that we have just described is called *orthographic visibility analysis*, since an orthographic imaging system is used. There are other kinds of imaging systems. The following sections describe visibility analysis for some other imaging systems.

A.2 Convex Imaging Systems

Standard computer graphics applications are based on the kinds of scenes that the human visual system perceives. The lens of the human eye is convex, and as a result *perspection* occurs - objects appear to shrink as they get farther away from the viewer. The shrinkage is linear, and can be modeled by the *convex imaging system* depicted in Figure A.2. In this system, the viewer is located at the vertex of a viewing cone. This vertex is called the *focal point* of the imaging system. Only surface regions inside the cone can be seen by the viewer. The lens has radius 1. The convexity of the lens determines the shape of the cone. The convexity can be measured by the *focal length* f, the distance from the focal point to the lens. Equivalently, the convexity can be measured by the viewing cone angle, which is equal to $\arctan(1/f)$.

Visibility analysis in this imaging system is called *convex visibility analysis*. First, we clip surface regions that are outside of the viewing cone. Then, we transform the objects of the system to orthographic coordinates by applying the *convex perspective transformation* to the vertices, as described below. Next, we perform hidden surface removal. The resulting visible surface regions can be transformed back to the original coordinate system by applying the inverse convex perspective transformation to the vertices.

The convex perspective transformation from a point (x, y, z) in eye coordinates to a point $(x_{new}, y_{new}, z_{new})$ in orthographic coordinates is given by the following equations [Sutherland et al 1974]:

$$x_{new} = \frac{fx}{-z} \tag{A.1}$$

$$y_{new} = \frac{fy}{-z} \tag{A.2}$$

$$z_{new} = \frac{(z+f)f}{z} \tag{A.3}$$

In Equation A.1, the term $x/(-z)$ causes the object to shrink linearly as it gets farther away from the viewer. (Recall that the viewing axis is along the negative z-axis.) Multiplication by f causes visible points to be transformed to the interval $[-1, 1]$ on the x-axis. Equation A.2 performs the same computation for the y-axis. Equation A.3 modifies the z-component of points to preserve straight lines and flat faces in the transformed coordinate system. This equation assumes that the original z-coordinate is negative. This assumption is valid since surfaces outside of the viewing cone are clipped before the transformation.

The convex perspective transformation can be inverted by the following equations:

$$z = \frac{f^2}{z_{new} - f} \tag{A.4}$$

$$y = \frac{-y_{new} z}{f} \tag{A.5}$$

$$x = \frac{-x_{new} z}{f} \tag{A.6}$$

We have extended our visibility algorithm to perform either orthographic or convex visibility analysis. We can summarize the algorithm as follows:

Algorithm A.1 *Visibility Analysis*

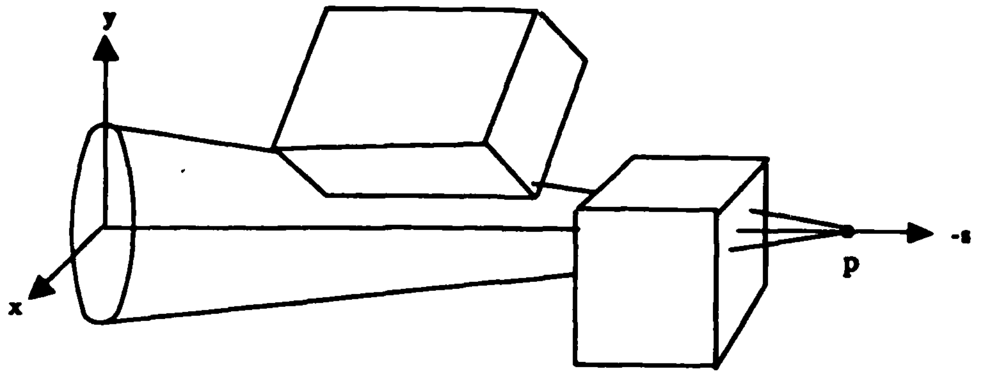


Figure A.3: A concave imaging system. The focal point is **p**. The viewer is located to the left of the lens.

1. *Transform the environment into eye coordinates.*
2. *Clip surface regions behind the viewer and outside of the viewing volume.*
3. *Transform the system into orthographic coordinates, if necessary, using perspective transformation.*
4. *Perform hidden surface removal.*
5. *Transform the visible surface regions back to perspective coordinates, if necessary, using inverse perspective transformation.*
6. *Transform the visible surface regions back to world coordinates.*
7. *Return the visible surface regions.*

A.3 Concave Imaging Systems

Consider the *concave imaging system* shown in Figure A.3. In this system, the viewer is located to the left behind a concave lens of radius 1. The concave lens causes objects to appear to grow linearly as they get farther from the viewer. Only surface regions inside an inverted cone containing the lens can be seen by the viewer. The vertex of the cone (the focal point of the imaging system) is the last point along the viewing axis that can be seen by the viewer. The focal length is defined as the distance along the viewing axis from the image plane to the focal point. A good example of a concave imaging system is a magnifying glass.

Visibility analysis in this imaging system is called *concave visibility analysis*. Algorithm A.1 suffices for this system, under the condition that surfaces behind the focal point must be clipped as well. Objects are converted to orthographic coordinates by the *concave perspective transformation*. The concave imaging system of Figure A.3 is similar to the convex imaging system of Figure A.2 except that the cone is inverted. In the convex imaging system, a point whose z-coordinate is z is located a distance $-z$ from the focal point. In the concave imaging system, the same

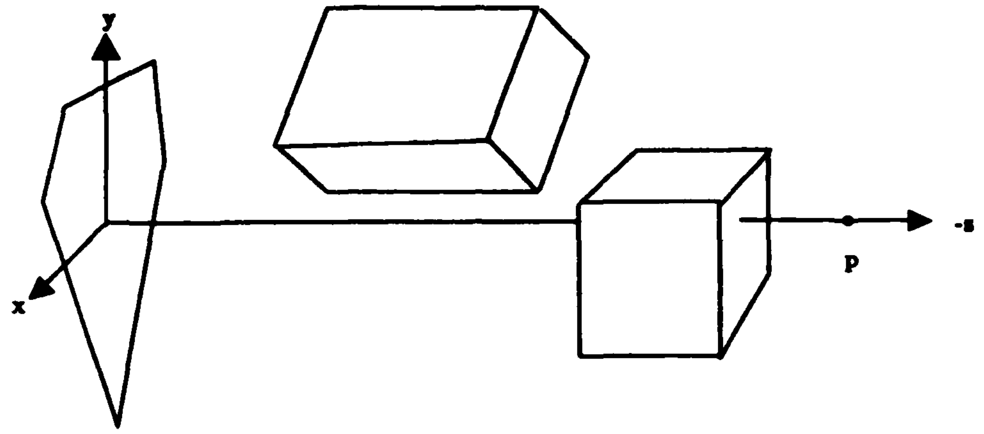


Figure A.4: A concave imaging system with a polygonal lens. The focal point is **p**. The viewer is located to the left of the lens.

point is located a distance $z + f$ from the focal point. Substituting $z + f$ for $-z$ in Equations A.1 - A.3 yields the equations for the concave perspective transformation:

$$x_{new} = \frac{fx}{z + f} \tag{A.7}$$

$$y_{new} = \frac{fy}{z + f} \tag{A.8}$$

$$z_{new} = \frac{zf}{z + f} \tag{A.9}$$

Concave perspective coordinates can be recovered from the orthographic coordinates of visible surface regions by applying the inverse concave perspective transformation:

$$z = \frac{f z_{new}}{f - z_{new}} \tag{A.10}$$

$$y = \frac{y_{new}(z + f)}{f} \tag{A.11}$$

$$x = \frac{x_{new}(z + f)}{f} \tag{A.12}$$

A.4 Polygonal Lens

We have generalized visibility analysis to handle orthographic, convex, and concave imaging systems. We can generalize it further by allowing an arbitrary polygonal lens. Consider a concave imaging system, for example. Suppose that a desired focal

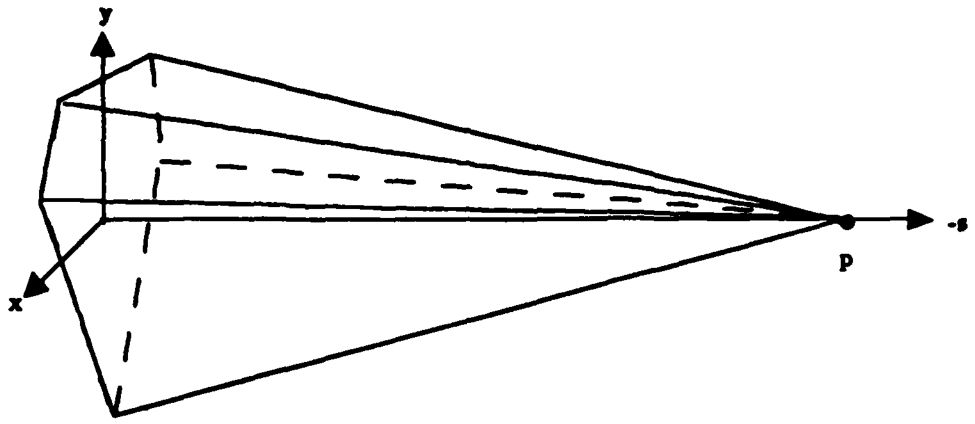


Figure A.5: The viewing volume of a concave imaging system with a polygonal lens is a pyramid.

point **p** is given, and that the lens is of a desired polygonal shape, as illustrated in Figure A.4. The surface regions that are visible in this imaging system are contained in a pyramid, as shown in Figure A.5. Visible surface regions in this system can be computed by concave visibility analysis. The choice of focal length is relatively unimportant. If it is desired that viewing coordinates be in the range [1,1], then the focal length can be set to the distance along the viewing axis from the focal point at which the maximum radius of the generalized cone is 1.

A.5 Hidden Surface Removal

Hidden surface removal involves clipping surface regions that are occluded by other surface regions. The input to the problem is a list of three-dimensional polygons, possibly concave, and a viewing axis. The imaging system for the problem is assumed to be orthographic.

We use the following simple algorithm:

Algorithm A.2 *Hidden Surface Removal*

1. *Sort the polygons along the viewing axis.*
2. *For each polygon, clip out all nearer polygons.*

This algorithm was chosen because it is conceptually simple, and because it returns 3-dimensional visible surface regions, not projections of them.

Step 2 can be implemented by polygon clipping, as described in Section A.6. This step takes $O(p^2)$ clipping operations in the worst case, where p is the number of input polygons. At best, the step can take $O(p)$ clipping operations. This occurs for example when one input polygon completely occludes all of the other input polygons. If the clipping is performed in closest-to-farthest order, then each occluded polygon disappears after only one clipping operation, and only $p - 1$ total clipping operations are required.

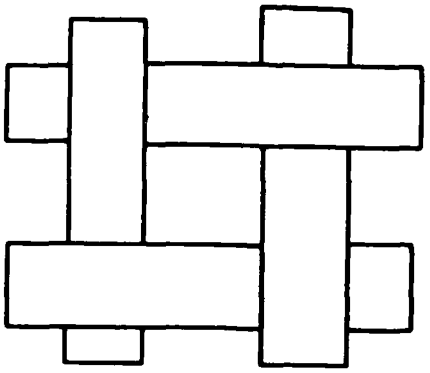


Figure A.6: A cycle under local or global occlusion.

A.5.1 Sorting

The sorting operation is performed using an *occlusion relation* on the input polygons. If polygon P_1 occludes P_2, then (P_1, P_2) is an ordered pair in the occlusion relation, and P_1 comes before P_2 in the polygon ordering. One simple occlusion relation is *local occlusion*. Given a particular viewing axis, polygon P_1 occludes P_2 if P_1 intersects P_2 in the xy plane, and is in front of P_2 with respect to the viewing axis.

One disadvantage of local occlusion is that it requires quadratic worst-case time in the number of edges, because of the polygon intersection. A more serious disadvantage is that the polygons must be sorted each time visibility analysis is to be performed on an environment. In many applications, such as the geometric simulation of Chapter 4, visibility analysis is performed many times on an environment. In this type of application, it would be desirable if a *global occlusion* relation could be computed for all of the visibility computations. In global occlusion, no viewing direction is assumed. A polygon P_1 occludes P_2 if for some viewing axis, P_1 occludes P_2. This relation can be computed by testing whether a vertex of P_1 is exterior to P_2, and a vertex of P_2 is interior to P_1. Passing the first test implies that a light ray can travel from P_1 to the visible side of P_2. Passing the second test ensures that P_1 is visible when the light ray penetrates it.

Let us examine the properties of local and global occlusion. In a numerical sort, the ordering relation $<$ is a total ordering. Neither local nor global occlusion are total orderings. Under local occlusion, there is no ordered pair in the relation corresponding to two polygons which do not intersect in the xy plane. In global occlusion, two polygons which are parallel and nonintersecting are not represented in the relation.

The lack of a total ordering means that we must perform a topological sort on the polygons. But a topological sort requires that the relation is a partial order, i.e. that it has no nontrivial cycles. Figure A.6 shows an environment that produces a cycle under local or global occlusion.

There is a very simple algorithm which eliminates cycles in all local occlusion relations. If all of the input polygons are split by the plane of all other input

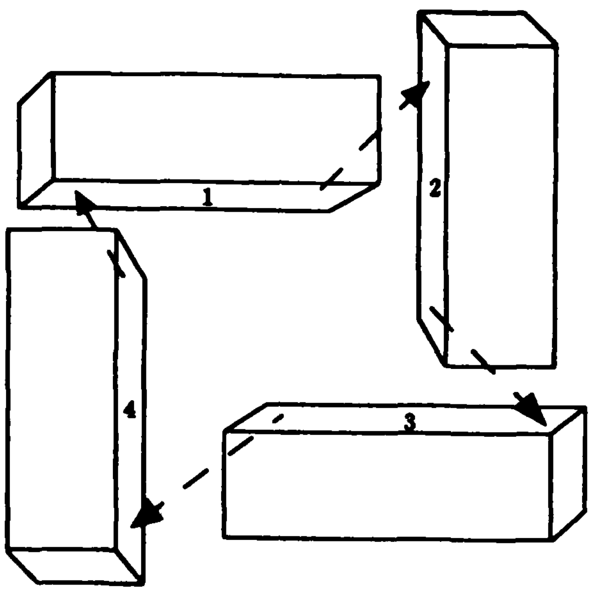


Figure A.7: A cycle using global occlusion. Face 1 can occlude face 2. Face 2 can occlude face 3. Face 3 can occlude face 4. Face 4 can occlude face 1.

polygons, then no cycles are possible under local occlusion. The reason is that after splitting, no polygon is both exterior and interior to another polygon. Thus, a light ray which leaves one polygon and penetrates another polygon cannot possibly return to the original polygon.

Unfortunately, there is no algorithm to eliminate cycles in a global occlusion relation. The reason is that a global occlusion relation is the set of all possible local occlusions. This allows cycles consisting of light rays with different directions, as shown in Figure A.7. This situation cannot really happen, because of the straight-line nature of light, but global occlusion doesn't know that. This means that it is not possible in general to construct a global occlusion relation that is a partial relation.

However, our algorithm doesn't give up completely on global occlusion. We define an environment as a set of *clusters*. A cluster is a set of polygons which (1) has no internal global occlusion cycles, and (2) is linearly separable from all other clusters. The no-cycle criterion ensures that the cluster can be topologically sorted. Since global occlusion is being used, this has to be done only once. The separability criterion ensures that a cluster can be treated as a single object with respect to occlusion.

An example of a cluster is a convex polyhedron. In a convex polyhedron, no face can be exterior to another face. No occlusions are possible, and thus no cycles are possible. Clusters are not limited to convex polyhedra, however. They must simply be consistent with the cluster criterion.

Our algorithm topologically sorts the polygons within clusters only once. Then, for each visibility computation, the clusters are topologically sorted. For a given viewing axis, we can compute whether cluster C_1 occludes C_2 by computing the projections of C_1 and C_2 onto the xy plane, and testing for intersection. If they intersect, and the viewer is on the same side of the separating plane between the clusters as C_1, then C_1 occludes C_2. Although this test is expensive, it is often avoided by performing a minimax test.

This clustering algorithm is based on ideas from [Schumacker et al 1969]. Although we didn't implement it, they present a faster way of sorting clusters. For a given environment, a decision tree is generated which computes the sorted list of clusters, given a viewpoint. The decision tree only has to be generated once, and can be evaluated in time $O(c)$, where c is the number of clusters. Our cluster sort is a topological sort, which requires making $O(c^2)$ occlusion computations, each of which may be expensive.

A.6 Polygon Clipping

Polygon clipping is ubiquitous in visibility analysis. Almost all of the problems discussed in this appendix can be implemented by some form of polygon clipping.

The purpose of polygon clipping is to clip sections from a polygon. Our algorithms must work on concave polygons, since the applications that we encounter in this thesis are not limited to convex polygons.

Here are some common clipping problems:

1. $Clip(P_1, P_2)$: Given two coplanar polygons P_1 and P_2, clip the shape of P_2 out of P_1. Either polygon may be concave.

2. $Cut(P, \mathbf{n}, \mathbf{q})$: Given a polygon P and a plane with unit normal vector $\mathbf{n}$ through point $\mathbf{q}$, clip out all portions of P that are interior to the plane.

3. $Cut(H, \mathbf{n}, \mathbf{q})$: Given a polyhedron H and a plane $(\mathbf{n}, \mathbf{q})$, clip out all portions of H that are interior to the plane.

The problem $Cut(H, \mathbf{n}, \mathbf{q})$ can be used to solve the visibility analysis problem of clipping all surface regions that are behind the viewer and outside of the visible volume. The polyhedron H represents an obstacle in the environment. The plane $(\mathbf{n}, \mathbf{q})$ bounds the visible volume. This problem can be reduced to the problem $Cut(P, \mathbf{n}, \mathbf{q})$ by calling $Cut(P_H, \mathbf{n}, \mathbf{q})$ for each face polygon P_H of H. An article by Sutherland and Hodgman [1974] describes an efficient algorithm for the problem $Cut(P, \mathbf{n}, \mathbf{q})$.

The problem $Clip(P_1, P_2)$ is used in hidden surface removal (Section A.5). The rest of this section presents an algorithm for this problem.

A.6.1 Minimax Test

The algorithm for $Clip(P_1, P_2)$ involves finding all intersections between edges of P_1 and edges of P_2. This operation dominates the running time of the algorithm.

A total of e_1e_2 edge intersections are called for, where e_1 is the number of edges in P_1, and e_2 is the number of edges in P_2. The rest of the algorithm is linear in e_1 and e_2. Therefore, it is a good idea to perform a *minimax* test on the polygons before intersecting them. The minimax test examines a common case in which the polygons clearly do not intersect. The test determines whether a horizontal or vertical line can be drawn between the polygons without intersecting either one. Our representation for polygons contains slots for the maximum and minimum X and Y values of the vertices. The polygons P_1 and P_2 do not intersect if any of the following conditions hold:

$$P_{1x}^{max} < P_{2x}^{min}$$
$$P_{2x}^{max} < P_{1x}^{min}$$
$$P_{1y}^{max} < P_{2y}^{min}$$
$$P_{2y}^{max} < P_{1y}^{min}$$

If none of the conditions hold, then the test is inconclusive, and the polygons may or may not intersect. However, the test can be performed in constant time, so it is well worth running before the rest of the algorithm. In practice, it eliminates a surprising number of cases from further consideration.

A.6.2 Edge Intersection and Labeling

The next step of the algorithm is to intersect edges of P_1 with edges of P_2. The intersection of two edges can lead to one of the following results:

1. They do not intersect.
2. They intersect at a point.
3. They overlap.

The information obtained by intersecting two edges that overlap is redundant, since edges which adjoin the overlapping edges provide point intersections which bound the intersection region. Figure A.8 illustrates this. Thus, we will concentrate on point intersections. When two edges intersect at a point, an *intersection vertex* is inserted into both polygons. The intersection vertices are connected to each other by a link. This connection will be used to trace out the clipped region from P_1.

The edge of P_1 which follows each intersection vertex is labeled for visibility at this time. Denote the original vertices of P_1 and P_2 before intersection as *corner vertices*. The interior angle at a non-corner intersection vertex is always equal to π. This fact allows us to quickly compute the visibility of edges which follow non-corner intersection vertices, as follows. The vertices of each polygon are stored in counter-clockwise order. By examining the sign of the cross product of two intersecting edges, it is possible to determine whether the edge of P_2 is crossing into or out of P_1. If it is crossing into P_1, then the next edge of P_1 is visible. If it is crossing out

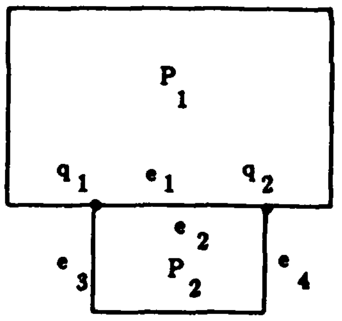


Figure A.8: Edges e_1 of P_1 and e_2 of P_2 overlap. The intersection points which bound the intersection region, $\mathbf{q}_1$ and $\mathbf{q}_2$, can be obtained by intersecting edges e_3 and e_4 with e_1.

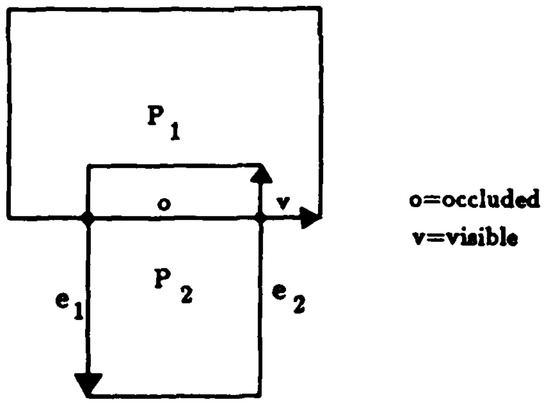


Figure A.9: Computation of edge visibility. By examining the sign of the cross product of two edges at a noncorner intersection vertex, we can determine whether the edge of P_2 is crossing into or out of P_1. If it is crossing into P_1, like $\bar{e}_2$ above, then the next edge of P_1 is visible. If it is crossing out of P_1, like $\bar{e}_1$ above, then the next edge of P_1 is occluded.

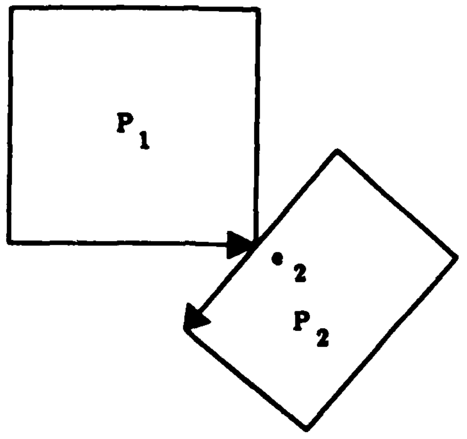


Figure A.10: An intersection involving a corner vertex of P_1. Edge e_2 appears to cross out of P_1, but the next edge of P_1 is not occluded.

of P_1, then the next edge of P_1 is occluded. Figure A.9 illustrates this computation. This computation is only guaranteed to work at non-corner intersection vertices. At these vertices, the previous and next edge are parallel in both polygons, and thus there is a clear crossing of the boundary of P_1.

Figure A.10 shows an intersection involving a corner vertex of P_1. Edge e_2 appears to cross out of P_1, but the next edge of P_1 is not occluded. Figure A.11 shows an intersection involving a corner vertex of P_2. Again, edge e_2 appears to cross out of P_1, but the next edge of P_1 is not occluded. At corner vertices, we can compute whether the next edge of P_1 is occluded by checking whether it is in the interior span of the previous and next edges of P_2.

After all intersection vertices have been computed, visibility labels are propagated along P_1 in the counter-clockwise direction to unlabeled edges (edges that are not preceded by an intersection vertex). For example, in Figure A.9, the visible label can be propagated along to the rest of the unlabeled edges of P_1.

There are a number of special cases to check for after edge intersection and labeling have been performed. If all edges of P_1 are occluded, then we return nothing. If all edges of P_1 are visible, then we return a copy of P_1. If no edge intersections are detected, then none of the edges of P_1 can be labeled, since the edge labeling scheme requires at least one label to propagate. In this case, one of the following statements is true:

1. P_1 is completely occluded by P_2.

2. P_1 is completely visible.

3. P_2 forms a hole inside P_1.

We can test for case 1 by picking a point of P_1, and testing for ***polygon containment*** in P_2. This can be done in linear time [Shamos 1975]; we use the linear

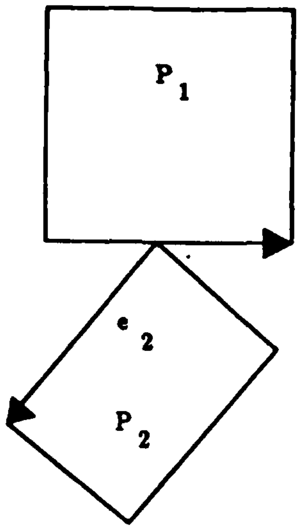


Figure A.11: An intersection involving a corner vertex of P_2. Edge e_2 appears to cross out of P_1, but the next edge of P_1 is not occluded.

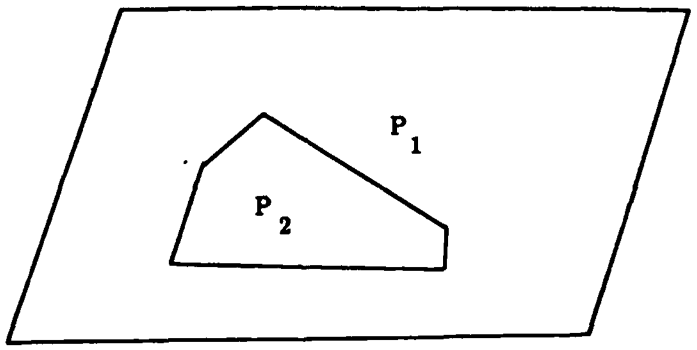


Figure A.12: Clipping P_2 from P_1 produces a hole in P_1.

algorithm presented by Sedgewick [Sedgewick 1983]. If case 1 holds, then we return nothing.

If case 1 does not hold, then we can test for case 2 by picking a point of P_2 and testing whether it is exterior to P_1. If case 2 holds, then we return a copy of P [illegible]

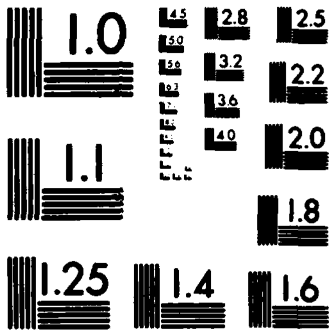
If case 2 does not hold, then we have to represent a hole in P_1.

A.6.3 Holes

One way to represent holes is to keep a list of hole polygons for each [illegible] chose not to use this representation because it forces all polyg[illegible] with holes. This makes polygon operations like interse[illegible] more difficult to implement.

Consider the hole created by clipping P_2 [illegible].

AD-A186 418 PLANNING AND TEACHING COMPLIANT MOTION STRATEGIES(U) 3/3
MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL
INTELLIGENCE LAB S J BUCKLEY JAN 87 AI-TR-936
UNCLASSIFIED N00014-85-K-0124 F/G 12/9 NL


1.0
4.5
5.0
5.6
6.3
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

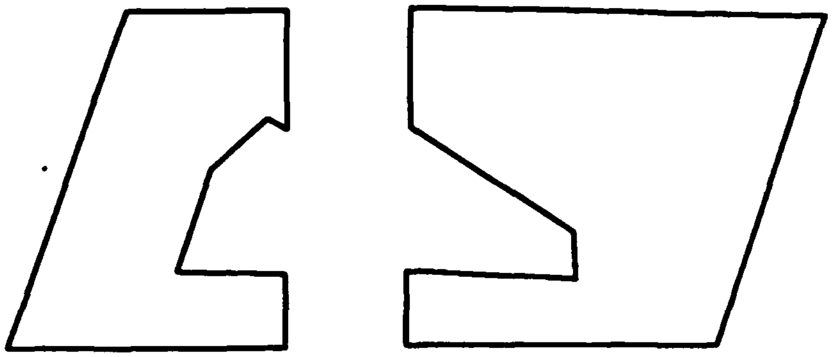


Figure A.13: The resulting polygons after clipping P_2 from P_1 in Figure A.12.

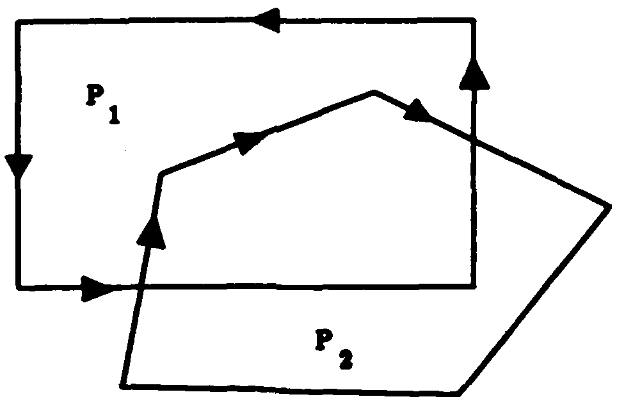


Figure A.14: The tracing algorithm. Starting at a visible edge of P_1, we traverse the boundary in the counter-clockwise direction until an intersection vertex is reached. At this point, we jump onto P_2, and traverse P_2 in the clockwise direction until another intersection vertex is reached. Then, we jump back onto P_1, and continue in the counter-clockwise direction until we return to the starting edge.

this hole by splitting P_1 in two and clipping P_2 out of the resulting halves. This results in the two concave polygons shown in Figure A.13.

A.6.4 Tracing

Now that all of the unusual cases have been taken care of, we are left to solve the case where P_1 and P_2 partially intersect. This case can be solved by a tracing algorithm [Weiler and Atherton 1977]. Starting at a visible edge of P_1, we traverse the boundary in the counter-clockwise direction until an intersection vertex is reached. At this point, we jump onto P_2, and traverse P_2 in the clockwise direction until another intersection vertex is reached. Then, we jump back onto P_1, and continue in the counter-clockwise direction. This process continues until we return to the starting edge. Figure A.14 shows an example of this computation.

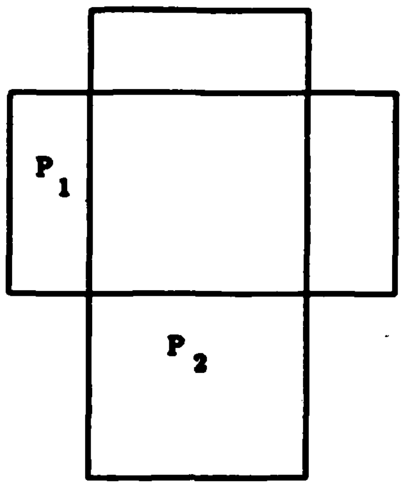


Figure A.15: A clipping application that produces multiple polygons.

A special case to watch out for while tracing is illustrated by Figures A.10 and A.11, in which a point contact occurs with an exterior polygon. This type of intersection vertex should be ignored. It only occurs when the intersection vertex is a corner vertex of P_1 or P_2. It can be avoided by making sure that the edge of P_2 that is to be followed is in the interior span of the previous and next edges of P_1.

The clipping operation may produce more than one resulting polygon. Figure A.15 shows an example of this. To obtain all of the resulting polygons, the tracing operation must be repeated until all visible edges of P_1 have been traversed.

Appendix B

Generalized Spring Trajectory Error and Overshoot

This chapter derives the maximum trajectory error and overshoot of a generalized spring trajectory, assuming perfect position sensing.

Suppose that the robot is initially in configuration s, and is commanded to go to configuration p. In its attempt to move directly from s to p, there may be a directional velocity error, bounded by θ_v. The robot is continuously being pulled towards p. This has a corrective effect on trajectories which attain maximal directional error. Figure 3.9 shows the possible trajectories that result from the corrective action. The trajectories are bounded by a collection of spiral curves.

Two parameters of generalised spring trajectories that a planner needs to know are the maximum trajectory error ϵ_{t_0} and the maximum overshoot ϵ_{o_0}, as shown in Figure 3.9. We will compute these parameters for the planar case. The results will hold in three dimensions as well.

Consider a trajectory on one of the bounding spiral curves. The extreme point of this trajectory in the y direction is the point where the maximum trajectory error ϵ_{t_0} occurs. The extreme point of this trajectory in the x direction is the point where the maximum overshoot ϵ_{o_0} occurs. We will use calculus to solve for these extrema.

Referring to Figure B.1, we place s at the origin of a planar coordinate system, and assign p the coordinates $(p, 0)^T$. The distance from s to p is p. Let $\mathbf{x} = (x, y)^T$ be a point on one of the bounding spiral curves. A vector in the direction of the commanded velocity from x is

$$\mathbf{v}_c = \mathbf{p} - \mathbf{x} = \begin{pmatrix} p - x \\ -y \end{pmatrix}. \tag{B.1}$$

An outward perpendicular vector to $\mathbf{v}_c$ with the same magnitude is

$$\mathbf{v}_\perp = \begin{pmatrix} -y \\ x - p \end{pmatrix}. \tag{B.2}$$

A vector in the direction of the instantaneous velocity of the boundary point x is

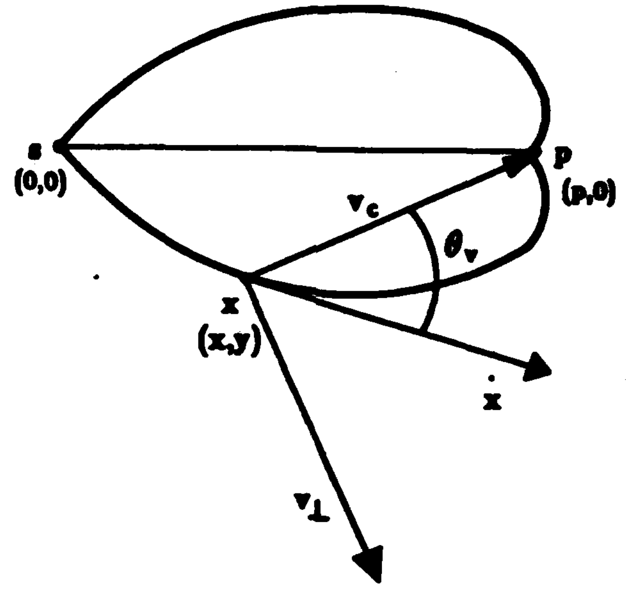


Figure B.1: **x** is a configuration on one of the spiral curves which bound generalised spring free space trajectories. $\mathbf{v}_c$ is a vector in the direction of the commanded velocity from **x**. $\dot{\mathbf{x}}$ is a vector in the direction of the instantaneous velocity at **x**.

$$\dot{\mathbf{x}} = \mathbf{v}_c + \mathbf{v}_\perp T = \begin{pmatrix} p - x - yT \\ -y + (x-p)T \end{pmatrix}, \tag{B.3}$$

where $T = \tan\theta_v$. The magnitude of $\dot{\mathbf{x}}$ doesn't affect the shape of the boundary trajectory. The linear system of Equation B.3 can be written in the form:

$$\dot{\mathbf{x}} = A\mathbf{x} + B \tag{B.4}$$

$$A = \begin{pmatrix} -1 & -T \\ T & -1 \end{pmatrix} \tag{B.5}$$

$$B = \begin{pmatrix} p \\ -pT \end{pmatrix} \tag{B.6}$$

We can solve this linear system for **x** in terms of p and T using standard linear system techniques (for example, see Luenberger [1979]). The general solution is equal to the convolution of the exponential matrix of A, e^{At}, with B:

$$\mathbf{x}(t) = \int_0^t e^{A(t-\tau)} B \, d\tau. \tag{B.7}$$

e^{At} can be computed by the equation

$$e^{At} = M e^{\Lambda t} M^{-1}, \tag{B.8}$$

where M is a matrix whose columns are eigenvectors of A, and Λ is a diagonal matrix with the eigenvalues of A on the diagonal. The eigenvalues of A are equal to

$$-1 \pm Ti. \tag{B.9}$$

Two eigenvectors of A are:

$$\begin{pmatrix} 1 \\ i \end{pmatrix}, \begin{pmatrix} 1 \\ -i \end{pmatrix} \tag{B.10}$$

We then obtain:

$$M = \begin{pmatrix} 1 & 1 \\ i & -i \end{pmatrix} \tag{B.11}$$

$$M^{-1} = \begin{pmatrix} \frac{1}{2} & \frac{-i}{2} \\ \frac{1}{2} & \frac{i}{2} \end{pmatrix} \tag{B.12}$$

$$e^{At} = \begin{pmatrix} e^{(-1-Ti)t} & 0 \\ 0 & e^{(-1+Ti)t} \end{pmatrix} \tag{B.13}$$

Substituting these values into Equation B.8, we obtain

$$e^{At} = \begin{pmatrix} \frac{(e^{2Tti}+1)e^{-Tti-t}}{2} & \frac{i(e^{2Tti}-1)e^{-Tti-t}}{2} \\ \frac{-i(e^{2Tti}-1)e^{-Tti-t}}{2} & \frac{(e^{2Tti}+1)e^{-Tti-t}}{2} \end{pmatrix}. \tag{B.14}$$

Plugging this into Equation B.7, and integrating, we obtain

$$\mathbf{x}(t) = \begin{pmatrix} p(1 - e^{-t}\cos(Tt)) \\ -pe^{-t}\sin(Tt) \end{pmatrix}. \tag{B.15}$$

The maximum trajectory error occurs when $\dot{y} = 0$. Substituting Equation B.15 into the bottom half of Equation B.3, and setting the result to 0, we obtain

$$pe^{-t}\sin(Tt) - pe^{-t}T\cos(Tt) = 0. \tag{B.16}$$

Thus, the maximum trajectory error occurs at time

$$t_t = \frac{\theta_v}{T}. \tag{B.17}$$

From Equations B.15 and B.17, the maximum trajectory error is

$$\epsilon_{t_0} = pe^{-t_t}\sin\theta_v. \tag{B.18}$$

The maximum overshoot occurs when $\dot{x} = 0$. Substituting Equation B.15 into the top half of Equation B.3, and setting the result to 0, we obtain

$$pe^{-t}\cos(Tt) + pe^{-t}T\sin(Tt) = 0. \tag{B.19}$$

This equation has multiple solutions, since the spiral encircles the commanded position infinitely. The maximum overshoot occurs the first time that $\dot{x} = 0$, which is given by

$$t_o = \frac{\theta_v + \frac{\pi}{2}}{\tan\theta_v}. \tag{B.20}$$

From Equations B.15 and B.20, the maximum overshoot is equal to

$$\epsilon_{\alpha_0} = p(1 + e^{-t_o}\sin\theta_v). \tag{B.21}$$

Appendix C

AML Program for Peg-in-Hole Insertion

This appendix contains a partial listing of an AML program to insert a rectangular peg into a rectangular hole. The subroutine SMOVE is a force feedback loop which implements a compliant motion towards a commanded position with a given stiffness. SDMOVE is like SMOVE except that the commanded position is given as an offset from the current position. CALIBRATE_HOLE picks up the peg from a corner of the hole, and computes the location and size of the hole, using the force sensors. This makes the program independent of the absolute location of the hole. PEG moves the robot to a start position above the hole, and inserts the peg into the hole, using a two-step compliant motion strategy.

```
XYZ: STATIC <JX,JY,JZ>;                          -- TRANSLATIONAL JOINTS
RPY: STATIC <JR,JP,JW>;                          -- ROTARY JOINTS
RPYHOME: STATIC <0.,0.,45.>;                     -- HOME ROTATION
STRAINS: STATIC <SLS,SLP,SLT,SRS,SRP,SRT>;       -- STRAIN GAUGES
BASE_STRAINS: STATIC 6 OF REAL;                  -- BASE READINGS
GUARD_FORCE: STATIC 100.;                        -- STOPS A GUARDED MOVE
STICK_FORCE: STATIC 3000.;                       -- INDICATES STICKING
GRASP_FORCE: STATIC 5000.;                       -- GRASPING FORCE
MOVEINC: STATIC .005;                            -- MOVE DISTANCE
NOM_STIFFNESS: STATIC 1000./MOVEINC;             -- NOMINAL STIFFNESS
--
-- PEG-IN-HOLE PARAMETERS
--
PEG_X_RADIUS: STATIC .5;
PEG_Y_RADIUS: STATIC .6;
HOLE_X_CLEARANCE: STATIC .025;
HOLE_Y_CLEARANCE: STATIC .025;
HOLE_HEIGHT: STATIC 1.25;
HOLE_LEFT: STATIC REAL;
HOLE_RIGHT: STATIC REAL;
HOLE_FRONT: STATIC REAL;
HOLE_BACK: STATIC REAL;
HOLE_BOTTOM: STATIC REAL;
HOLE_TOP: STATIC REAL;
```

```
SMOVE: SUBR(P,
            FMAX DEFAULT GUARD_FORCE,
            K DEFAULT NOM_STIFFNESS);
--
-- PURPOSE: PERFORMS A GENERALIZED SPRING COMPLIANT MOTION
--          IN WHICH THE REACTION FORCE F = K(X-P), WHERE P
--          IS THE COMMANDED POSITION, X IS THE ACTUAL POSITION,
--          AND K IS A VECTOR OF STIFFNESS CONSTANTS. THE
--          MOTION IS TERMINATED WHEN THE REACTION FORCE
--          IN THE DIRECTION AGAINST MOTION EXCEEDS FMAX.
-- INPUT: P = COMMANDED POSITION (3 OF REAL)
--        FMAX = MINIMUM TERMINATING FORCE (REAL, OPT)
--        K = STIFFNESS VECTOR (3 OF REAL, OR JUST ONE REAL, OPT)
--
  S: NEW QGOAL(XYZ);
  PNEXT: NEW S;
  NSEGS: NEW INT IS MAG(P-S)/MOVEINC;
  V: NEW (P-S)/NSEGS;
  FDIR: NEW -V/MAG(V);
  I: NEW 0;
  F: NEW FORCES;
  FOLD: NEW 3 OF 0.0;
  WHILE (I LT NSEGS) AND
        (DOT(FOLD,FDIR) LT FMAX) OR (DOT(F,FDIR) LT FMAX)
  DO BEGIN
    I = I+1;
    PNEXT = PNEXT+V;
    DISPLAY('POS=',QGOAL(XYZ),' F=',DOT(F,FDIR),EOL);
    MOVE(XYZ,PNEXT+F/K);
    FOLD = F;
    F = FORCES;
    -- ON TRANSITION FROM CONTACT TO FREE SPACE,
    -- REPLAN THE INTERPOLATED TRAJECTORY.
    IF (I LT NSEGS) AND (MAG(FOLD) GT GUARD_FORCE)
                    AND (MAG(F) LE GUARD_FORCE)
    THEN BEGIN
      S = PNEXT = QGOAL(XYZ);
      NSEGS = MAG(P-S)/MOVEINC;
      V = (P-S)/NSEGS;
      FDIR = -V/MAG(V);
      I = 0;
      END;
    END;
 IF I LT NSEGS THEN
   DISPLAY('MOTION TERMINATED BY FORCE ',DOT(F,FDIR),EOL);
 END;
```

```
SDMOVE: SUBR(DP,
               FMAX DEFAULT GUARD_FORCE,
               K DEFAULT NOM_STIFFNESS);
--
-- PURPOSE: PERFORMS A GENERALIZED STIFFNESS COMPLIANT MOTION
--          TO A DIFFERENTIAL COMMANDED POSITION.
-- INPUT: DP = OFFSET OF COMMANDED POSITION (3 OF REAL)
--        FMAX = MINIMUM TERMINATING FORCE (REAL, OPT)
--        K = STIFFNESS VECTOR (3 OF REAL, OR JUST ONE REAL, OPT)
--
 SMOVE(QGOAL(XYZ)+DP,FMAX,K);
 END;
```

```
PEG: SUBR(X1,Y1,Z1,X2,Y2,Z2);
--
-- PURPOSE: INSERTS PEG IN HOLE.
-- INPUT: X1 = DISTANCE BEHIND HOLE FOR FIRST MOTION
--        Y1 = LATERAL OFFSET OF FIRST MOTION
--        Z1 = DEPTH BELOW TOP OF HOLE FOR FIRST MOTION
--        X2 = X OFFSET FROM HOLE CENTER FOR SECOND MOTION
--        Y2 = Y OFFSET FROM HOLE CENTER FOR SECOND MOTION
--        Z2 = DEPTH BENEATH HOLE BOTTOM FOR SECOND MOTION
--
 -- MOVE ARM TO START POSITION.
 DISPLAY('MOVE ARM TO CLEAR POSITION',EOL);
 GUIDE(ARM);
 MOVE(RPY,RPYHOME);
 MOVE((JX,JY),(HOLE_FRONT+PEG_X_RADIUS,(HOLE_LEFT+HOLE_RIGHT)/2.));
 MOVE(JZ,HOLE_TOP+.1);
 DELAY(1.0);
 SET_BASE_STRAINS;
 SDMOVE((0.,0.,-1.),GUARD_FORCE);
 -- COMPLIANT MOTION TO BACK OF HOLE.
 SMOVE((HOLE_BACK-X1,
        (HOLE_LEFT+HOLE_RIGHT)/2.+Y1,
        HOLE_TOP-Z1),
       STICK_FORCE);
 -- COMPLIANT MOTION TO BOTTOM OF HOLE.
 SMOVE(((HOLE_FRONT+HOLE_BACK)/2.+X2,
        (HOLE_LEFT+HOLE_RIGHT)/2.+Y2,
        HOLE_BOTTOM-Z2),
       STICK_FORCE);
 END;
```

```
FORCES: SUBR;
--
-- PURPOSE: COMPUTES XYZ FORCES FROM STRAIN GAUGES.
-- OUTPUT: FORCE VECTOR (3 OF REAL)
--
  STR: NEW SENSIO(STRAINS,6 OF REAL)-BASE_STRAINS;
  RETURN((-STR(2)+STR(5),-STR(1)-STR(4),STR(3)+STR(6)));
  END;

SET_BASE_STRAINS: SUBR;
--
-- PURPOSE: SETS THE BASE STRAIN GAUGE READINGS.
--
  RETURN(BASE_STRAINS=SENSIO(STRAINS,6 OF REAL));
  END;
--
CALIBRATE_HOLE: SUBR;
--
-- PURPOSE: CALIBRATES ABSOLUTE HOLE LOCATIONS.
--
  -- GRASP PEG WHILE IT IS IN THE HOLE.
  DISPLAY('GUIDE HAND TO CLEAR POSITION.',EOL);
  GUIDE(ARM);
  MOVE(RPY,RPYHOME);
  MOVE(JG,2.0*PEG_X_RADIUS+1.0);
  DISPLAY('GUIDE HAND TO GRASPING POSITION',EOL);
  GUIDE(XYZ);
  CGRASP(0.,GRASP_FORCE);
  HOLE_BOTTOM = QGOAL(JZ);
  -- MEASURE THE LOCATION OF THE HOLE.
  DMOVE(JZ,HOLE_HEIGHT/2.);
  DISPLAY('ADJUST PEG IF DESIRED.',EOL);
  GUIDE(JG);
  SET_BASE_STRAINS;
  SDMOVE((0.,2.5*HOLE_Y_CLEARANCE,0.),GUARD_FORCE);
  HOLE_RIGHT = QGOAL(JY);
  DMOVE(JY,-HOLE_Y_CLEARANCE);
  SDMOVE((0.,-1.5*HOLE_Y_CLEARANCE,0.),GUARD_FORCE);
  HOLE_LEFT = QGOAL(JY);
  MOVE(JY,(HOLE_LEFT+HOLE_RIGHT)/2.);
  SDMOVE((-2.5*HOLE_X_CLEARANCE,0.,0.),GUARD_FORCE);
  HOLE_BACK = QGOAL(JX);
  DMOVE(JX,HOLE_X_CLEARANCE);
  SDMOVE((1.5*HOLE_X_CLEARANCE,0.,0.),GUARD_FORCE);
  HOLE_FRONT = QGOAL(JX);
  MOVE(JX,(HOLE_FRONT+HOLE_BACK)/2.);
  -- MOVE TO THE START POSITION.
  MOVE(JZ,HOLE_BOTTOM+HOLE_HEIGHT+.2);
  MOVE(JX,HOLE_FRONT+PEG_X_RADIUS);
  DELAY(1.0);
  SET_BASE_STRAINS;
  SDMOVE((0.,0.,-1.5),GUARD_FORCE);
  HOLE_TOP = QGOAL(JZ);
  END;
```

END

DATE

FILMED

DEC.

1987